A comparison of languages which operationalise and formalise {KADS} models of expertise

Fensel, D.; van Harmelen, F.A.H.

published in
Knowledge Engineering Review
1994

DOI (link to publisher)
10.1017/S0269888900006767

document version
Publisher's PDF, also known as Version of record

The Knowledge Engineering Review, Vol. 9, pp. 105-146, 1994

A Comparison of Languages which Operationalize and Formalise KADS Models of Expertise

Dieter Fensel (*) and Frank van Harmelen (+)

(*) Institut für Angewandte Informatik und Formale Beschreibungsverfahren (AIFB)
University of Karlsruhe, 76128 Karlsruhe, Germany
tel: +49-721/6084754, fax: +49-721/693717
e-mail: fensel@aifb.uni-karlsruhe.de

(+)Department of Social Science Informatics, SWI
University of Amsterdam, Roeterstraat 15, NL-1018 WB Amsterdam, The Netherlands
tel: +31-20/5256121, fax: +31-20/5256896
e-mail: frankh@swi.psy.uva.nl

Abstract.

In the field of Knowledge Engineering, dissatisfaction with the *rapid-prototyping* approach has led to a number of more principled methodologies for the construction of knowledge-based systems. Instead of immediately implementing the gathered and interpreted knowledge in a given implementation formalism according to the rapid-prototyping approach, many such methodologies centre around the notion of a conceptual model: an abstract, implementation independent description of the relevant problem solving expertise. A conceptual model should describe the task which is solved by the system and the knowledge which is required by it. Although such conceptual models have often been formulated in an informal way, recent years have seen the advent of formal and operational languages to describe such conceptual models more precisely, and operationally as a means for model evaluation. In this paper, we study a number of such formal and operational languages for specifying conceptual models. In order to enable a meaningful comparison of such languages, we focus on languages which are all aimed at the same underlying conceptual model, namely that from the KADS method for building KBS. We describe eight formal languages for KADS models of expertise, and compare these languages with respect to their modelling primitives, their semantics, their implementations and their applications. Future research issues in the area of formal and operational specification languages for KBS are identified as the result of studying these languages. The paper also contains an extensive bibliography of research in this area.

The Knowledge Engineering Review, Vol. 9, pp. 105-146, 1994

Introduction

One reaction to the so-called “software crisis” in the late sixties was the development of well-structured process models, tools, and methods in the domain of software engineering which should allow the construction of large, reliable, and maintable computer programs. During the last years, significant effort has been spent to develop methodological foundations for the construction of knowledge-based systems [DKS93]. Whereas most of this work aims at tool support for specific subtasks of the development process, the KADS-I project ([WSB92], [SWB93]) aimed at a general framework and methodological support for most of phases of the development process of knowledge-based system.

One of the major results of the KADS-I project is the introduction of a life cycle oriented methodology for developing expert systems, including a proposal for a so-called “model of expertise” as the result of the knowledge acquisition phase. This model of expertise is based on the identification of different types of knowledge, which are distinguished as four different layers within the model of expertise. Accordingly, different modelling primitives have been proposed to capture the different types of knowledge that are represented on the various layers of a model of expertise. However, all these proposed modelling primitives have only been defined informally. Thus, they lack a clear semantics and do not provide a basis for formally analysing the modelled expertise or for evaluating it by executing the model. To remedy this deficiency, a number of languages have recently been proposed in the literature that describe KADS models more precisely and formally. Three different kinds of languages can be identified:

- Languages which formalize KADS models of expertise,
- languages which operationalize KADS models of expertise, and
- languages which both formalize and operationalize KADS models of expertise.

Until now, the following languages have been discussed in the literature:

- FORKADS ([Wet90], [WeS91], [Wet92])
- KARL ([FAL91], [Ang93], [Fen93b], [AFS94])
- $K_{BS}SF$ ([JoS92], [VJS93])
- $(ML)^2$ ([BaA92b], [HaB92], [AHS93])
- Model-K ([KVS91], [KaV92], [Kar93], [KaV93])
- Momo ([LKV92], [VoV93], [VVW93])
- OMOS ([Lin92a], [Lin92b], [Lin93])
- QIL ([ARS92], [AKS93]).

In addition, [Möl92] proposes KL-ONE as an operationalization and formalization language. In a more elaborated way, this idea is realized by the language MODEL/KADS [Bar93]. [Gei92] defines an operational language in the domain of decision support systems which can also be viewed as an operational knowledge specification language because it is based on the KADS model of expertise. A further knowledge specification language which is not oriented at the KADS model of expertise is DESIRE ([KoT90], [LPT92], [LPT93]). A KBS is described via a set of interacting modules that each include an object-layer and a meta-layer. A logical language is provided for describing each of these layers.

In this paper, we describe the eight languages mentioned above, and analyse their differences and commonalities. To allow a meaningful comparison, we restricted ourselves to knowledge specification languages which are oriented at the KADS model of expertise (thereby ignoring languages like DESIRE) and from this subset of languages we chose the most prominent

approaches, i.e. approaches which can report some applications.

The paper is organized as follows. The first section shows why there is a need for such languages. These inquiries are used in the second section to develop criteria which can be used to analyse the different languages. The third section discusses eight languages in detail using these criteria. For every language we discuss their central idea and their main properties. A survey is given in the appendix. The last section classifies and compares the different languages.

Chapter 1
Why Do We Need Formal Specification Languages for Knowledge-Based Systems?

1.1 The Necessity of Model-Based Knowledge Engineering

Traditional Software Engineering (SE) has long since recognised the value of two fundamental premises that should underlie the construction of complex software systems:

- A clear distinction should be made between the functional specification of a software system on the one hand, and its design and implementation on the other.
- Functional specifications should as far as possible be given in a precise, unambiguous and preferably formal language that abstracts from implementation details.

Knowledge Engineering (KE) is the branch of SE which deals with the construction of Knowledge Based Systems (KBS). As KE matures into a separate discipline, these two fundamental premises of Software Engineering are increasingly acknowledged to be of similar value to Knowledge Engineering. This has lead to a diminishing enthusiasm for so called rapid prototyping as an approach to KE. *Rapid Prototyping* means immediately implementing the acquired knowledge in a computer program. The running system may then be used to evaluate the gathered knowledge. This method for constructing a system reflects the cyclic nature of the modelling process. The running system provides immediate feedback to the knowledge acquisition process for revision or completion. For some time this method was therefore applied, in most cases, to construct an expert system. Nevertheless, this method has some well-known disadvantages (cf. [AFS90]):

- The knowledge engineer who implements the system has to perform many tasks at the same time. He has to analyse the given information, as well as to design, implement, and evaluate the system.
- Different knowledge aspects have to be considered and implemented simultaneously and are therefore mixed.
- The running system is the only documentation of expertise.
- The view of knowledge is determined by the chosen implementation language, and implementational details become mixed with aspects of the conceptual model.

This gave rise to the study of more principled methodologies for KBS construction. Central to such more principled methodologies has been the first of the two premises above: the separation of the functional specification of a KBS from its technical design and implementation. What in SE is traditionally called a *functional specification* has received a

number of names in KE: a knowledge level description (after Newell's distinction between knowledge and symbol level [New82]), a *conceptual model*, or a *model of expertise*. Such an abstract, implementation independent description of the functionality of a KBS has become the basis of a number of KBS development methodologies (e.g. [AFL93], [ChJ93], [MDK92], [PET92], [Ste92], [WSB92]).

Some care must be taken in this analogy between *functional specification* from SE and *conceptual model* or *model of expertise* from KE. Traditionally, a functional specification is taken to be a description of the I/O behaviour of the system without any reference to *how* this behaviour should be realised. In KE, however, a large amount of an expert is concerned with the *how*, and this knowledge must be represented in the conceptual model. We must therefore distinguish between the *how at the knowledge level* (the *how* which is concerned with knowledge in order to solve a task effectively and efficiently), and the *how at the symbol level* (*how* to implement effective and efficient algorithmic solutions of the specified problem-solving method). The central idea of a model of expertise is then to specify the *what* of a system (it's I/O behaviour) and the *knowledge level how*, but to abstract from the symbol level aspects [SAW89].[1]

1.2 The Necessity of Formal and Operational Specification Languages

Since the construction of such model of expertise plays a central role in a number of KBS methodologies, it is of prime importance that such a conceptual model can be formulated clearly and unambiguously, as stated in the second of the two general SE principles given above. The role of formal languages in the construction of a functional specification is widely accepted in SE, and we only briefly summarise the well-known arguments in favour of the formalisation of functional specifications:

- It reduces the vagueness and ambiguity of natural language descriptions by adding an additional level of preciseness and uniqueness.
- Formal descriptions narrow the cognitive gap between a mental model of a system and its implementation and can be used as a base line to verify the implemented system.
- A formalised specification allows for validation of completeness and consistency, either through formal proofs, or through symbolic execution.
- A formalised specification can be mapped to an operational one, which allows prototyping. An operational language integrates the flavour of model evaluation by prototyping into a well-structured development process [Flo84].

Such formal or operational descriptions should not replace informal specifications but they can be used to add an additional level of refinement to an informal or semiformal description.

In contrast with the long history of formal functional specifications in traditional SE, research on formalising conceptual models in knowledge engineering has only recently received wider attention. This development is witnessed by a number of recent publications and activities such as the ECAI'92 Workshop on Formal Specification Methods for Complex Reasoning Systems [TrW93].

This paper contributes to the research on specification languages for KBS by describing, analysing and comparing a number of such languages. All the languages that we discuss aim

1. The term *knowledge level* caused much confused debate in the AI community. In the context of this paper, no more should be read in this term than that a *knowledge-level representation* is an abstract representation of a system that captures the functionality of such a system without defining how such functionality should be computationally realised. Knowledge level does not mean that there are no symbols, but that there are the right (non-machine oriented) symbols (cf. [Fen93a]).

at formalising/operationalizing a particular type of conceptual model, namely as proposed by the KADS model of expertise. The fact that all the languages that we discuss in this paper share the basic structure of the conceptual model they aim to formalise enables a meaningful comparison of these languages.

Our choice for languages based on KADS model of expertise is not an arbitrary one. KADS has been an influential methodology in research on KE, as witnessed by the large number of groups (both in industry and in academy) who exploit KADS in the construction of KBS [SWB93]. The widespread interest in KADS has also lead to a number of recent proposals for competing formal languages all aiming at formalising KADS models of expertise, which makes the comparison effort of this paper a timely one.

1.3 Principles of the KADS-I Project

In order to keep this paper self contained, we first give a brief description of the essential aspects of KADS and especially of the model of expertise, because this model forms the common framework for all the languages discussed in this paper.

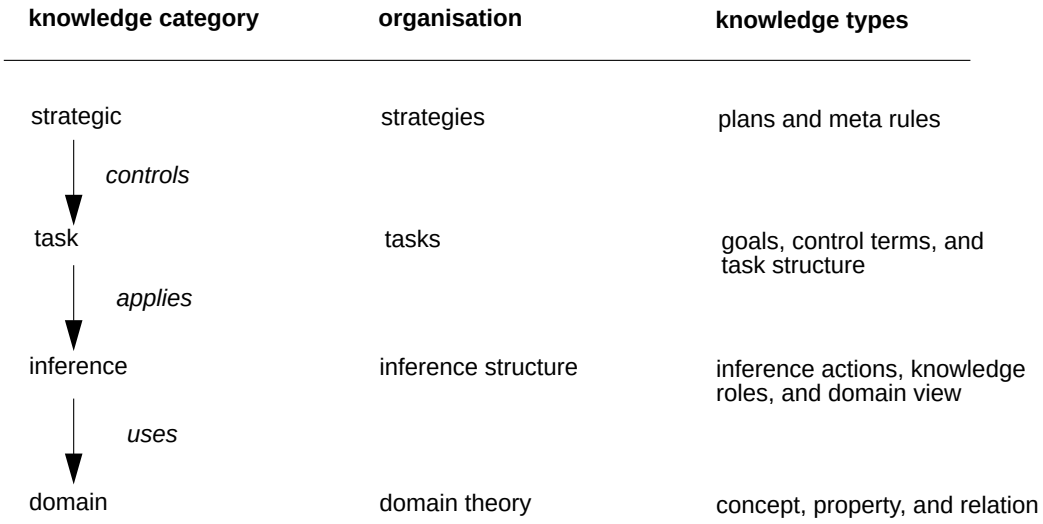

Figure 1. The four-layer model of expertise [WSB92].

Modelling Expertise. The significant distinction between conventional software systems and knowledge-based systems is that the latter explicitly represent the significant amount of knowledge which is necessary to solve the desired task effectively and efficiently. A very important part of the KADS methodology is therefore the model of expertise which describes the different kinds of knowledge required to solve the given tasks. The model of expertise distinguishes different types of knowledge, it defines primitives to express them, and organizes them into several layers. The model of expertise distinguishes static knowledge and three types of control knowledge. The goal of a model of expertise is to provide a model of the problem solving behaviour required of a knowledge-based system in an implementation independent way. These models consist of four hierarchically organised layers and prescribe the contents of the layers and the relations among them, as follows (see figure 1).

- *Domain layer:* This is the *lowest* of the four layers, and represents knowledge about the application domain of the system. An important property of the domain layer is that the knowledge should be represented in a way that is as independent as possible from the way it will be used (i.e. the domain layer is a *declarative, relatively task-neutral representation*

of the domain knowledge of a system). It has two main purposes. First, it should define a conceptualization of the domain. Second, it should define a declarative theory of the domain which must provide all the domain knowledge required to solve the given tasks. All further layers of the model of expertise contain knowledge which *control* the use of the domain knowledge. The modelling primitives for the domain layer are concepts, properties/values, relations, and structures.

- *Inference layer:* This second layer defines the first type of control knowledge. It specifies the inferences that constitute a problem-solving method and specifies how to *use* the knowledge from the domain layer for these inferences. This is done in two ways: the inference layer specifies the *basic inference steps* that can be made using the domain knowledge, and the *knowledge roles* model the premises and conclusions of the inferences.
  The inference steps are assumed to be elementary in the sense that they are completely described by their names, an input/output specification and a reference to the domain knowledge that they use. "*The actual way in which the inference is carried out is assumed to be irrelevant for the purposes of modelling expertise*" [WSB92]. The inference layer also specifies the data-dependencies between the steps and roles. These dependencies are specified in a network of inference actions and knowledge roles known as an *inference structure*. The inference layer *restricts* the use of the domain layer knowledge and *abstracts* from it. It restricts all possible inferences to the set of inferences which are defined at the inference layer. This is done to improve the efficiency of the problem-solving process. The inference layer abstracts from the domain layer by using task-specific names for inferences and roles (e.g., patient data are referred as observables and diagnoses are referred as hypotheses as shown in figure 2). The domain-independent formulation of the inference layer should support its reuse, i.e. its application for similar tasks in different application domains. The connection of a domain and inference layer is defined by the domain view which defines a relation between an inference step and the domain knowledge it uses and by the connection of the knowledge roles and the domain entities which correspond to them. Although the inference layer specifies the basic inference steps, it does *not* specify any control knowledge defining their ordering: no ordering is imposed on the various inference steps.
- *Task layer:* A task represents a fixed strategy for achieving problem solving goals. The purpose of the task layer is to specify *control* over the execution of the basic inference steps specified at the inference layer. It does this by imposing an ordering on these steps in terms of execution sequences, iterations, conditional statements etc. "In KADS, tasks only refer to inferences and not explicitly to domain knowledge" [WSB92]. For each task there is a task structure, which hierarchically refines a given task by subtasks and elementary steps, i.e. inference actions. In addition, the control flow between the sub-steps which refine the task is defined.
- *Strategy layer:* This highest of the four levels in a KADS models is concerned with *task selection*: how to choose between various tasks that achieve the same goal.

The separation of domain knowledge on the one hand and knowledge which controls its task-specific use on the other hand enables two kinds of reuse. Domain knowledge can be used for several tasks and inference/task layers can be used for several domains. The following figure 2 shows a possible model of expertise of heuristic classification[2] (cf. [Cla85]). The inference layer contains the elementary inference steps and knowledge roles of this problem-solving method. Observables are abstracted to more abstract descriptions of the case data, these abstract descriptions are matched with hypothesis classes, and these classes are refined until a

2. The example does not include a strategy layer.

final solution has been found. The control flow between these inferences is defined at the task layer and the domain knowledge which is used by these generic inferences is defined at the domain layer.

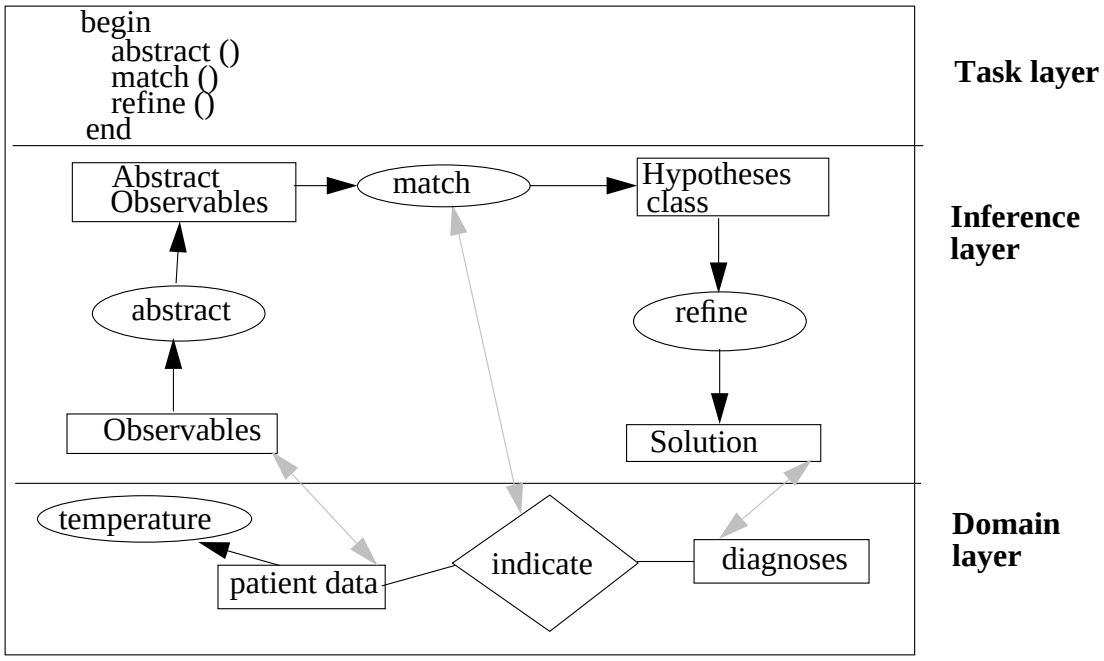

Figure 2. An example for a model of expertise.

The model of expertise should not be used to model the cooperation of the systems with its environment.[3] This aspect is clearly separated and modelled in a separate model which is called the *model of cooperation* [GrB92]. Therefore, most of the languages which are discussed in the following do not aim at representing the cooperation of the specified system with further agents. As a result, they do not aim at representing the user-system-cooperation.

For a more detailed description of KADS, we refer to [WSB92].

Chapter 2
How to Analyse Formal and/or Operational Knowledge Specification Languages?

For analysing the various languages, we require comparison criteria. The definition of these criteria is important since it will clearly influence the result of our investigation. What are the relevant criteria to investigate languages used to specify a model of expertise? Criteria which can be applied to compare programming languages seem hardly suitable, since such languages have entirely different purposes. Programming languages should allow an effective and efficient realisation of the modelled expertise at the symbol level, and should therefore deal with other aspects than the specification languages we are studying here.

As stated above, we are studying languages which allow a *formal* and/or *operational*

3. “The KADS model of expertise can be viewed as an autistic problem solver” [GrB92].

specification of models of expertise *at the knowledge level*. Therefore, three different kinds of questions seem relevant in the comparison of these languages [Bra79]:

- *Epistemological view:* From an epistemological viewpoint, we are interested in what the modelling primitives are that a language offers. As argued in the previous section, our comparison is made possible by the fact that all languages are based on a single underlying conceptual framework, namely the KADS model of expertise. Thus, at this level, we are specifically interested in (i) what the modelling primitives of a language are for each part of a KADS model; (ii) how close these primitives approach the KADS models as described in the literature, and (iii) whether the language in any way extends the prescriptions given by KADS models.
- *Formal view:* In a second set of comparison criteria, we will compare languages as mathematical constructs, and we will address questions concerning formal expressive power, semantics, types of formal constructions employed in the language, etc.
- *Operational view:* In a final set of criteria, we will view languages as means of operationalizing models of expertise. Questions concerning executability, computational power and efficiency will be of main concern in this view on the various languages.

The following subsections discuss a list of comparison criteria which are organised according to these three views.

2.1 The Epistemological View

The epistemological view most strongly exploits the fact that all languages in our comparison are based on KADS models. The correspondences in their underlying conceptual models make it possible to compare the languages on the basis of their modelling primitives, and allow us to structure our epistemological comparison criteria in a very specific way. For each of the major components of a KADS model (the layers and their connections) we will ask three types of questions:

- *Type 1: What* are the modelling primitives of a language for this part of a KADS model?
- *Type 2*: How closely do these modelling primitives *approximate* the properties of this part of a KADS model (as described in e.g. [WSB92])?
- *Type 3:* How much do the modelling primitives of a language *extend or modify* the structure, contents or properties of this part of a KADS model?

While the type 1 criteria are purely descriptive in nature, the type 2 and 3 criteria are more normative. However, we do not want to suggest that any deviation from the theory of KADS models is necessarily negative. In fact, we would expect that the formal investigations of KADS models through these languages would indeed lead to modifications of the KADS framework.

Clearly, the criteria in this epistemological view on specification languages are very KADS specific (and more so than the criteria from the other views). However, we claim that our comparison has general value because (i) KADS models are widely used in the European KBS community; (ii) KADS models embody general principles which also feature in other approaches to KBS modelling (albeit possibly in a different form) and (iii) the organisation of our criteria may serve as an example for similar comparison work based on other conceptual models.

Criteria for the *domain layer*[4]

- **D1 (Type 1):** What are the language primitives to express the domain conceptualisation

and the domain theory. Are they rich in the sense that they allow the expression of different kinds of knowledge by different language primitives? In particular, how does the language represent concepts, properties, values, relations and structures?

- **D2 (Type 2):** Can domain knowledge be expressed independently from its use? Is it free from control knowledge?
- **D3 (Type 2):** Does the language distinguish between domain-specific knowledge and case-specific data?
- **D4 (Type 3):** Are there any assumptions about the domain which determine the kinds of knowledge that can and cannot be modelled (e.g. change over time, monotonicity, uncertainty)?
- **D5 (Type 3):** Does the domain layer only act as a storage for domain knowledge, or does is also have a set of legal inferences that can be made with this knowledge?
- **D6 (Type 3):** Does the language offer any constructions for modularisation or decomposition of the domain knowledge?

Criteria for the *inference layer*

- **I1 (Type 1):** How are primitive inference actions represented? How is their input/output behaviour described?
- **I2 (Type 1):** How are the knowledge roles represented (i.e. the case-specific inputs and outputs of the inference actions)?
- **I3 (Type 1):** How is the domain view (i.e. access to static domain knowledge) represented?
- **I4 (Type 2):** Is the inference layer generic, i.e. is it free from domain specific expressions?
- **I5 (Type 2):** Are inference actions free from control knowledge, i.e. can the input/output behaviour be described without reference to a particular computation strategy?
- **I6 (Type 2):** Is the inference structure free from control knowledge? Can (or must) any control knowledge be modelled through the dependencies described in the inference structure?
- **I7 (Type 2):** Can inference actions communicate in any other way then through their connection knowledge roles?
- **I8 (Type 2):** Does a language have a predefined set of inference actions, and if so, how are they organized?
- **I9 (Type 3):** Are inference actions deterministic or non-deterministic (i.e. are they functions or relations)?
- **I10 (Type 3):** Does the language offer any constructions for hierarchical decomposition at the inference layer?

Criteria for the *task layer*

- **T1 (Type 1):** What primitives are used to represent control knowledge?
- **T2 (Type 1):** How are task-structures and subtask-decomposition represented?
- **T3 (Type 1):** How are the different states of the problem-solving process represented?
- **T4 (Type 1):** What types of conditional expressions can be used at the task-layer?
- **T5 (Type 2):** Is the formulation of task knowledge domain independent (i.e. free from references to the domain layer)?

---

4. We number each of our criteria, and we will use these numbers to refer to these criteria in later sections.

- **T6 (Type 3):** Can non-determinism be expressed at the task layer?

Criteria for the *connections*

- **C1 (Type 1):** How is the connection between inference and domain layer represented?
- **C2 (Type 1):** How is the connection between task and inference layer represented?
- **C3 (Type 2):** Do the connections enable the reusability of the layers? (Can a domain layer be re-used by multiple inference layers; can an inference layer be re-used for multiple domain layers; can an inference layer be re-used under multiple task-layers)

Because the *strategic layer* is rather underspecified in the KADS literature, and because many of the languages discussed in this paper do not deal with the strategic layer (we suspect for the very reason of its unspecificity), we will not include the strategic layer in our comparison. Unless stated otherwise, languages do not have any provisions for representing a strategic layer.

2.2 The Formal View

When we view languages as purely formal constructions (that is: independently from the fact that they are intended to specify or operationalize KADS models), we can compare the languages on the basis of the following general criteria:

- **F1:** Is the language *aimed at formalising* KADS models of expertise (i.e. aiming at precision and disambiguity, and allowing formal reasoning about models without regard to the computational aspects of such a formalism)?
- **F2:** What are the *mathematical constructs* that form the basis of the language? Of particular interest will be the constructs used to represent *dynamic behaviour.*
- **F3:** Has a declarative *semantics* been defined for the language, and if so, what is the nature of this semantics?
- **F4:** What is the formal *expressive power* of the language? Note that expressive power in this formal sense is very different from the epistemic modelling power discussed in the previous set of criteria. For example, predicate calculus and modal logic have equal formal expressive power, since each can be translated into the other (by an embedding in one direction and by reification in the other), but notwithstanding this formal equivalence, the epistemic power of the languages is very different: modal logic introduces concepts and distinctions which disappear when translated into predicate calculus.

2.3 The Operational View

When viewed as a computational mechanism (in order to make a model of expertise executable), the following general criteria apply to a language:

- **O1:** Is the language *aimed* at operationalizing KADS models of expertise?
- **O2:** What is the *computational paradigm* of the language (e.g. logical, functional, procedural)?
- **O3:** Has an *operational semantics* been defined for the language, and if so, what is its relation with the declarative semantics?
- **O4:** What is the *computational power* of these constructs? In general, this computational power will determine the *effective computability* of the language. (Note that this

computational power is again different from both the formal and the epistemological expressive power discussed above)

- **O5:** How *efficiently* can the language be implemented? This is related both to the computational paradigm and to the computational power of the language.

An additional question could have been whether there are *tools* which support the application of the languages like editors, or whether there are tools which check modelled expertise for consistency, correctness or completeness. We deleted this topic from our comparison because of space limitation and because the developed tools are not an inherent feature of the languages. For each language we will briefly report on any tool development that has taken place, but this is not done as part of our systematic comparison because a serious discussion of this topic would require a paper on its own. Similarly, we will briefly mention reported applications for each language, without systematically comparing these (although, of course, their use in applications is ultimately the most important criterion in the long run).

Chapter 3
The Languages in More Detail

> *"Come, let us go down, and there confound their language, that they may not understand one another's speech!" [Genesis, 11:7]*

This section will discuss in detail each of the languages addressed in our comparison. Because our comparison criteria are divided into three viewpoints, and to facilitate the application of these criteria, we will describe each language from these three points of view: epistemological, formal and operational. In section 4 it will turn out useful to group the languages on the basis of their main purpose, which can be either the *operationalization* of a KADS model, the *formalisation* of such a model, or *both* of these. In section 4 we will order the languages on a continuum from operational to formal, and this ordering will be one of the major explanatory factors for many of the commonalities and differences between the languages. In this section, we will anticipate that ordering of the languages by presenting the languages from most operational to most formal.

3.1 Operational Models of Problem-Solving (OMOS)

Operational Models of Problem-Solving (OMOS) ([Lin92a], [Lin92b], [Lin93]) is a language to operationalize KADS models of expertise. It was developed at the German National Research Center for Computer Science, GMD, in Bonn.

The main point of OMOS is to extend the Role-Limiting Methods [Mar88] or Method-to-Task approach [Mus89] by integrating them into a KADS framework. The Role-Limiting Methods approach provides shells with corresponding knowledge acquisition tools like MOLE, MORE or SALT [Mar88]. The shells contain a fixed data structure and an algorithm which works on it. The knowledge acquisition process consists of two activities. First, a proper shell must be chosen, and second, this shell must be filled with cases, i.e. with assertional knowledge.

The approach is not meaningful if no previously developed shell fits the chosen task. Therefore, OMOS allows a bottom-up development of such problem-solving methods and

provides some tools which support the acquisition of assertional knowledge. A problem-solving process can be modelled by using role and value changes of given instances. Besides extending existing expert shell approaches OMOS is based on the KADS model of expertise and therefore makes it possible to evaluate such models using explorative prototyping.

3.1.1 The Epistemological View

OMOS is implemented on top of BABYLON [CPV89]. BABYLON provides frames, rules, Prolog, and constraints and is implemented in Common Lisp. OMOS exploits frames and Prolog for the domain layer, and rules for the inference layer

The Domain-Layer Primitives

At the *domain layer* terminological knowledge can be modelled by concept hierarchies and relationships between concepts. Frames available in BABYLON are used to model concepts. These frames can form a hierarchy with multiple inheritance. Relationships between concepts are defined by a Prolog predicate with the corresponding concepts as argument types. Assertional knowledge can be modelled at the domain layer using instances of concepts and relations between them. This can be done extensionally by enumerating the elements of a relation or intensionally by using Prolog clauses. OMOS requires finite extensions of relations.

The Inference-Layer Primitives

As explained, the inference layer contains inference actions and roles as modelling primitives. The functionality of an inference action in OMOS is defined by stating:

- the type of value-assignment it implements (no *value-assignment*, an *initialization*, or a *modification* of an existing value),
- its type of role-assignment (*no change*, *transfer* a domain instance from one role to another, *assign* it to a new role without undoing its old role-assignment, or *substituting* a new concept for a concept that previously played that role),
- whether the inference action queries the user for missing knowledge or not [5],
- the domain-layer relation used by the inference action,
- and the input, output, and control knowledge roles.

Because each of these properties has a finite number of values, there are a finite number of inference actions that can be built by combining these properties. A compiler translates these definitions into a forward rule-scheme.

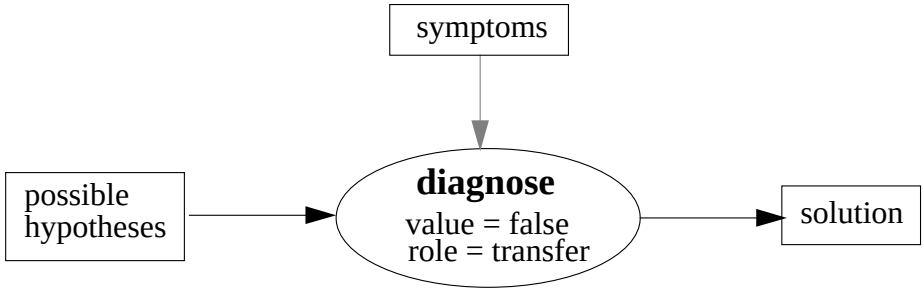

Figure 3. An inference action in OMOS.

5. Because the type of inference actions indicates whether they include user interaction or not, part of the user/system interaction may be modelled by OMOS (cf. chapter one).

The inference which is drawn by an inference action is described by the domain relation it uses. Therefore a high amount of the problem-solving knowledge must be represented as tuples of domain relations. It is not possible to represent problem-solving knowledge generically by describing the inference which is drawn by an inference action. Only the interaction of the inference actions can be represented generically through network of control, input, and output knowledge roles.

A knowledge role can be associated with instances of concepts of the domain layer and contain input, control information, or output of inference actions. The execution of an inference action can have two possible consequences: it can change the attribute values of instances of its input knowledge role and it can assign instances (probably with changed attribute values) of its input knowledge role to its output knowledge role. In addition, the domain layer is used as a global store. All inference actions can read it and write on it.

An inference action has exactly one input and one output role and several control roles. Only instances of the input role can be transformed to the output role. The members of a control role only control the inference but cannot be removed by it.

A small example of an inference layer in OMOS is given in figure 3. The inference action *diagnose* has the *possible hypotheses* as an input role, the *symptoms* as a control role and inferred *solution* as output role. Its value-assignment is *false* because there is no value change. Its role-assignment is *transfer* or *assign* because an element of *possible hypotheses* is inferred as a new element of *solution*. In the first case, the element would be deleted from *possible hypotheses*. A domain relation must specify the relationship between possible hypotheses and the control role *symptoms*. The domain relation could contain this knowledge encoded as tuples where the first element is a hypothesis and the second one is a symptom. Then the role *solution* is filled with all elements of the input role which are the first element of a tuple having the second element as an element of the control role *symptoms*.

The Task-Layer Primitives

The names of the inference actions define primitive actions at the task layer. These primitive actions can be combined by sequence, branch, and loops.

The Primitives Connecting the Layers

A knowledge role at the inference layer can be connected with a domain concept while an inference action must be connected with a domain relation. To define a mapping of an inference action, one must specify the corresponding domain relation and the correspondence between input and control knowledge roles of the inference action and the arguments of the domain relation. This mapping is defined through a direct naming of the domain-layer relations and arguments.

For the mapping between inference and task layer, inference actions return a boolean value indicating whether a change took place or not. In addition, every knowledge role can be evaluated to a boolean value (to indicate if the knowledge role is empty or not).

3.1.2 The Formal and Operational View

OMOS is not discussed as a formal specification language, but rather as an operationalization language. Therefore, no formal semantics for the modelling primitives is given.

OMOS is implemented using BABYLON, and therefore, BABYLON’s environment is available to OMOS. In addition, three tools for knowledge level analysis of OMOS

specifications were developed, which check completeness of domain concepts and domain relations (in combination with their use at the inference layer). These tools are straightforward because they use the 1:1 mapping of domain layer and inference layer and the finite extensions of concepts and relations at the domain layer.

3.1.3 Applications and Discussion

OMOS has been applied to a number of problems with a finite problem space, such as *planning* (the remodelling of parts of ONCOCIN [LiM92], in particular skeletal-plan-refinement); s*election* (selecting clamping tools for lathe turning [KLS91]), and a*ssignment* (the so-called Office-Plan or Sisyphus Problem, consisting of searching for an assignment of employees to appropriate offices under given constraints [Lin91], [Lin92e]).

The application of OMOS is limited to problems with a predefined problem space because no new instances of domain concepts or domain relations can be created during the problem-solving process. Therefore, only problems requiring selection and combination among a finite and predefined set of alternatives can be solved.

The intermediate results of the problem-solving process are changes in the value of domain elements and changes in their knowledge roles, but no traces of this are available to guide the further problem-solving process and no elements can be deleted or created during the problem-solving process. It is therefore difficult to model a problem-solving process which investigates several alternative paths.

OMOS provides restricted modelling primitives for representing problem-solving knowledge and its connection with domain knowledge. Every inference action has exactly one input and one output knowledge role. It is not possible to describe the inference drawn by an inference action in a generic manner, because inference actions have to refer directly to relations and instances at the domain layer.

3.2 MODEL-K

MODEL-K ([KVS91], [KaV92], [Kar93], [KaV93]) is a language for operationalizing KADS models of expertise. It has been developed at the German National Research Center for Computer Science, GMD in Bonn. It is again implemented in BABYLON [CPV89].

The main point of MODEL-K is the use of an existing AI shell, i.e. BABYLON, to operationalize KADS models of expertise. This is done by introducing KADS-specific modelling primitives in BABYLON. An operational model can then be built by attaching BABYLON code to these modelling terms. Similar to OMOS, MODEL-K allows the evaluation of the model of expertise by prototyping. Additionally, MODEL-K is considered as a language for the implementation of a final expert system. The incremental development of an expert system which reflects the conceptual structure of the knowledge level description of the expertise becomes possible. The conceptual structure thus becomes a tool for improving understandability and maintenance.

Another important feature of MODEL-K is its possibility of modelling reflective problem solving. This can be regarded as a step in the direction of a strategic layer.

3.2.1 The Epistemological View

MODEL-K consists of two sub languages (cf. figure 4). MODEL-K provides a *specification language* which allows an informal description of the expertise. In addition, MODEL-K provides an *operationalization language* which can be used to operationalize the terms defined by the specification language.

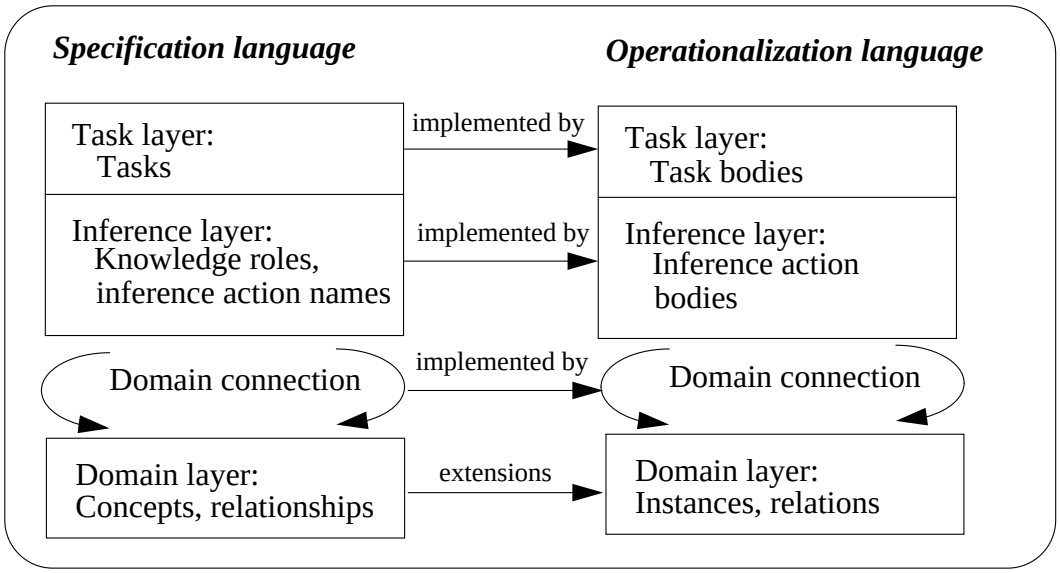

Figure 4. The two sub-languages of MODEL-K

The Domain-Layer Primitives

The specification language can be used to model terminological knowledge by concept hierarchies with inheritance and relationships between concepts. Frames of BABYLON are used to model concepts and relationships.

Assertional knowledge can be modelled with the operationalization language by instances of concepts and tuples which are elements of relations. These relations can be described extensionally or intensionally.

The Inference-Layer Primitives

The specification language can be used to define names of inference actions and their input and output knowledge roles. The external behaviour of an inference action is defined as well as the whole inference structure, but the way in which an inference action draws an inference is not specified: actions are regarded as black boxes.

The operationalization language allows the operational refinement of inference actions by adding an inference-action body. This body implements the behaviour of inference actions using BABYLON code, including Lisp, Prolog, constraints, etc.

The Task-Layer Primitives

The specification language can be used to describe tasks and their hierarchical refinement. The operationalization language adds task bodies allowing the specification of control knowledge. It provides a procedural language consisting of sequence, branch, and loops.

The Strategic-Layer Primitives

MODEL-K is one of the few languages with a strategic layer. This layer is used to model

reflective problem-solving. A model of expertise containing reflective modules consists of three different levels (see figure 5):

- An object system is modelled which consists of domain layer, inference layer, and task layer.
- One or more reflective modules are specified, which again consist of domain layer, inference layer, and task layer. The difference is the fact that their domain layers contain the model of the object system.
- A scheduler defines the control flow between the reflective modules.

An example of reflective problem-solving is given in [KaV93]. The reflective modules control the required run-time of the object-system, and try to decompose complex problems into subproblems which can each be solved efficiently by the object-system. An object system should assign employees to places in consideration of several constraints (i.e., the Sisyphusproblem [Lin92c]). The reflective problem-solver then deals with the following questions:

- One reflective module enables the user to restrict the maximal number of seconds to be spent and the maximum number of solutions to be found by the object-system.
- If the input problem is too complex it might be useful to decompose this problem into a sequence of increasingly complex sub-problems and solve those stepwise.

The modelling primitives for reflective problem-solving do not increase the expressive power of the modelled problem-solver. However, they allow modelling reflective problem-solving at the conceptual level and therefore support understandability and reusability.

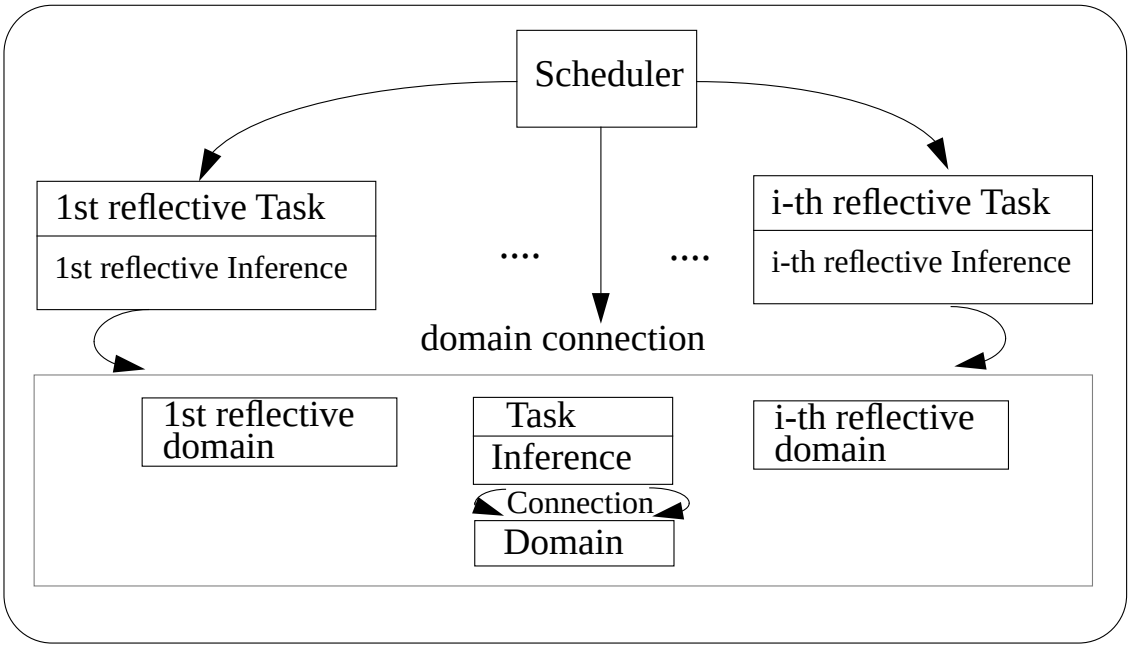

Figure 5. Modelling reflective problem-solving with MODEL-K

The Primitives Connecting the Layers

The names that the specification language introduces to express knowledge roles at the inference layer can be regarded as formal names which are associated in the operationalization language with their actual values at the domain layer. This association is realised by expressing the relation in BABYLON code.

As the knowledge roles are modelled as containers, two update commands are given. A knowledge role can be filled with domain knowledge, and the contents of a knowledge role can be written back to the domain layer.

The connection between task and inference layer is realised by using the names of inference actions as primitive subtasks in the task-decomposition. The condition in branches and loops at the task layer are arbitrary BABYLON expressions which can refer to the domain layer and to the contents of the knowledge roles.

3.2.2 The Formal and Operational view

MODEL-K is intended as an operationalization language, and not as a formal specification language. The knowledge model receives an operational semantics if the inference action and task bodies are implemented in BABYLON. The operational semantics is defined by the BABYLON interpreter.

MODEL-K provides the expressive power of programming languages like Lisp and Prolog. Special tools for supporting knowledge formalization, evaluation, assessment, or error correction are not discussed. MODEL-K contains the environment of BABYLON.

3.2.3 Applications and Discussion

MODEL-K has been applied to *assignment problems*, *namely* the Sisyphus problem ([VKS91], [DKV92]) and the assignment of airplanes to gates [VoK93]. In addition, *reflective modules* have been developed in the context of the Sisyphus problem [KaV93]: a case-based reasoner, a scheduler for the management of computing time, and several modules which transform or simplify problem descriptions.

MODEL-K uses its specification language to describe the structure of a model of expertise, and uses BABYLON to operationalize this model. In this way, MODEL-K attempts to smooth the transition from knowledge-level description to symbol-level implementation. MODEL-K is intended for implementing running expert systems while preserving the structure of the model of expertise. However, because of its use of BABYLON to express the bodies of inference actions, MODEL-K requires the user to make quite specific implementation commitments in the description of such actions.

3.3 MoMo

MoMo ([LKV92], [VoV93], [VVW93]) is a language to operationalize KADS models of expertise. It is developed at the German National Research Center for Computer Science, GMD in Bonn and is the successor of OMOS and MODEL-K. MoMo aims to extend the limited expressiveness of OMOS, and to restrict the unlimited computational power of MODEL-K. The degree of elaboration of a model depends on what is considered as essential. MoMo aims at allowing models at any desired grainsize. The parts that are unspecified can either be “executed” interactively by the user, e.g. for testing partial models, or they can be attached to available pieces of code reachable from its implementation environment CLOS (Common Lisp Object System, [Kee89]).

3.3.1 The Epistemological View

MoMo relies on Petri-nets for the inference layer, and leaves the choice of the domain-layer language open. The task layer provides the usual procedural constructs sequence, branch, and loop.

The Domain-Layer Primitives

MoMo does not come with a fixed domain layer language. It aims to allow the use of any language that is reachable from the CLOS environment. As a default, a language with concepts, instances, relations and rules is provided. No use of domain languages other than the default has been reported

The Inference-Layer Primitives

MoMo's inference layer is an extension to coloured Petri-nets. Petri-net places are used to model knowledge-roles, and consist of strongly typed multi-sets. Petri-net actions model primitive inference actions. On their input arcs, these actions can restrict the types of their input places and express further preconditions to the action by guard-predicates. The output arcs are labelled with expressions composed of procedures acting upon the input variables. Thus, an inference action is modelled by an action described by (i) variable names which label the arcs from the input places, (ii) a guard predicate which describes the precondition of the action, and (iii) arcs to the output places, labelled by terms built from input variables and operations on them. An inference structure behaves like a Petri-net: if from the input places elements can be matched that satisfy the type restrictions and the guard predicates, the action is enabled. If the action is executed, these elements are removed, and elements as specified by the output terms are moved into the output places.

As an extension to this standard Petri-net interpretation, MoMo provides variables that match the entire multiset of the input places, arbitrary subsets, or subsets of specified size.

The Task-Layer Primitives

The Petri-net specified at the inference layer is often non-deterministic, and allows a large number of possible execution paths (at any moment more than one action might be enabled). Rather than restriction this non-determinism by extending the net with control-flow actions (as would be the usual approach in Petri-nets), MoMo instead enforces additional control on the network through separate expressions at the task-layer. In these control expressions, primitive operations refer to inference actions, and these can be combined by sequence, branch and loop instructions.

The Primitives Connecting the Layers

The types, places and procedures of the inference layer must be connected to the domain layer. Such connections can be made either by a renaming operation (if there is a direct match with a domain layer construct), or the connect can be programmed in Lisp. This type of connection is necessarily much less restricted than in the other languages, because MoMo aims at using arbitrary domain-layer languages.

The connection between inference and task layer is through the identification of primitive tasks with inference actions. Furthermore, inference actions return a boolean value and predicates can be defined over the contents of knowledge roles.

3.3.2 The Formal and Operational View

MoMo was not conceived as verification language for KADS models. Hence, little attention has been paid to defining its semantics formally. However, being an extended coloured Petri-net, the inference layer has the usual Petri-net semantics. The task layer could be formally defined as a restriction on the non-determinism of the inference layer, but this has not been elaborated.

The open ended choice of the domain layer precludes the definition of a fixed semantics for this layer, as well as for its connection with the inference layer and therefore for the entire specification.

For its operational semantics and its execution, MoMo relies on CLOS. Visual editing and simulation tools are provided.

3.3.3 Application and Discussion

In [LKV92] the selection of clamping tools for lathe turning and in [VVW93] the scheduling problem of the ECAI'92 Workshop on Formal Specification Methods for Complex Reasoning Systems [TrW93] is sketched.

MoMo allows the integration of prototyping into the modelling process. By doing this, MoMo tries to integrate several principles of software engineering like graphical representation of modelling primitives, object-orientation, or typing.

Coloured Petri-nets are used in MoMo to represent the inference structure. This is a nonstandard use of such nets, since inference layers should not represent any control, but such knowledge could in principle be expressed in Petri-nets. This makes it hard to decide where in a model to express certain knowledge. The expressive power of the pre- and post-conditions of the coloured Petri-nets allows the inference layer of MoMo to specify all the control which should normally be specified at the task layer.

MoMo aims to remove the restrictions imposed by OMOS' limitations on the type of inference actions, and MODEL-K's heavy reliance on the underlying implementation language for defining inference action bodies. MoMo does indeed extend the restricted expressive power and flexibility of OMOS. However, it is not really clear how MoMo restricts MODEL-K (MoMo's second rationale) since the operations which label the output arcs of an action can be arbitrary complex programs.

3.4 FORKADS

FORKADS has been developed at the IBM Germany Scientific Center in Heidelberg. It was one of the first published approaches to formal KADS models ([Wet90], [WeS91], [Wet92]). Contrary to the languages described so far, FORKADS aims not only at operationalizing KADS models, but also at giving formal foundations to KADS models. Furthermore, the aim is to use this language as a (or perhaps even the only) communication medium between the people responsible for knowledge acquisition and those responsible for system design. For this purpose, the main foundation of FORKADS is a first-order logical language which is extended with notions of concept heterarchies and procedural attachment. In many respects FORKADS is rooted in the L_{LILOG} language [HeR91].

3.4.1 The Epistemological View

FORKADS is based on order-sorted logic with certain extensions, based on LILOG.

The Domain-Layer Primitives

In FORKADS, concepts are represented as sorts, and instances of a concept as constants of that sort. Sorts (concepts) can have a lattice-like subsumption relation to model subconcepts. So-called features and roles of concepts are represented by one-place functions and two-place relations over sorts. Other relations are modelled as n-place predicates.

The features mentioned above represent the general *domain model* of a FORKADS domain layer. Properties of individual instances of sorts, or the truth conditions on relations, roles and features are specified by a set of first-order axioms.

The Inference-Layer Primitives

FORKADS is the only language which has a predefined and fixed set of primitive inference steps. These primitive inferences are represented as predicates, and non-primitive inferences can be modelled as logical sentences built from these primitive inferences. Two types of primitive inferences are distinguished at the inference layer: *terminological* and *referential* inferences. The terminological inferences deal with properties represented in the sort-structure. Examples of such terminological inferences are to assign a sort to a variable, to create a constant of a given sort, and to bind a variable to a direct subsort of a given sort. These are rather non-standard operations in a logical language. They are based on the LILOG language, where they were introduced for describing the semantics of natural language.

The referential inferences infer properties of specific individuals. The main primitive referential inferences are to prove a goal (possibly establishing a value for a variable) and to test if a variable has a value.

These primitive inference actions are used to build non-primitive inferences. Each non-primitive inference is modelled as a logical sentence using primitive inferences, and has a so-called “inference-head”, which consists of the name of the inference and its parameters which can be used for calling the inference.

The Task-Layer Primitives

As with inferences, tasks, and subtasks are also expressed by means of a task-head and a task-body. The task-head contains name and parameters and can be used to call the (sub)task.

Bodies of tasks and subtasks are a sequence of statements which consist of calls to other (sub)tasks or inferences (which are regarded as primitive tasks). Furthermore, task bodies can contain loop-statements (for iterating over all elements in a sort) and conditional statements. Two other types of statements are allowed in task bodies, and these are applied to affect the set of axioms used for proving inferences: it is possible to either restrict or enlarge the set of axioms in the knowledge-base used for proving a goal.

The Primitives Connecting the Layers

FORKADS distinguishes between a specific domain model and a generic *stereotype*. Both are sort structures with their associated relations, but a domain model describes a specific application domain, while a stereotype describes a generic structure. Such stereotypes are used to define domain-independent inference structures and task layers (interpretation models). The connection between a specific domain model and the generic sort model of a stereotype is made by statements of the form *role_of(.,.)*, which link sorts and relations of the domain model to their counterparts in the sort model of the stereotype. FORKADS requires that this mapping

is a homomorphism between the two sort models when both are interpreted as graphs.

Since inferences are regarded as primitive tasks, and no further separation is made between task and inference layer, no separate mechanism is required to connect the task and inference layers.

3.4.2 The Formal and Operational View

The domain layer of FORKADS consists of an order-sorted, first-order predicate calculus. It is the only layer in FORKADS with a declarative semantics. The inference layer and task layers are only given a operational semantics in terms of an abstract interpreter. However, the properties of these semantics are hard to assess since their construction is along rather unconventional lines.

[BüW92] sketch an implementation of FORKADS consisting of a graphical modelling component GCONMOD and a code generator LCONMOD. LCONMOD generates Prolog code which is used to execute FORKADS inferences. It can use procedural attachments to predicates implemented in C and uses the inference machine of the knowledge representation language L_{LILOG} [HeR91]. The graphical tool GCONMOD allows editing and browsing of FORKADS structures either in terms of formulae, or in terms of a graphical representation of the dependencies in a FORKADS model.

3.4.3 Application and Discussion

FORKADS has been used in a commercial banking application project [Wet92].

FORKADS closely follows the usual KADS interpretation of domain layers in terms of instances and concepts, representing them by well-understood notions such as constants and sorts. The procedural form of the task layer is different from that of some of the other languages (e.g. the iteration over all elements of a sort), but otherwise contains common notions such as sequence, branching, and subroutining.

FORKADS' most striking features appear in the inference layer. The referential inference constructs like testing if a variable has a value and terminological ones like creating a new constant, as well as the possibility for procedural attachment require a nonstandard semantics which is considerably different from a standard semantics, thereby reducing the value of the semantic account. After all, one of the aims of a formal semantics is to clarify one formalism by describing it in terms of another, better known formalism.

3.5 KARL

The *Knowledge Acquisition and Representation Language (KARL)* ([FAL91], [Ang93], [Fen93b], [AFS94]) is a language used to formalize and operationalize KADS models of expertise and is developed in the context of the MIKE approach (Model Based and Incremental Knowledge Engineering) [AFL93]. A formal description of the expertise is automatically mapped to an operational one. KARL utilizes results of software engineering and information system design. The domain layer of KARL applies ideas of *semantic* and *object-oriented data models* ([Bee90], [ElN89]) to represent the static knowledge. The inference layer applies ideas

of *structured analysis* [You89] and the task layer language is influenced by languages like *RSL* [Alf90].

3.5.1 The Epistemological View

KARL contains the two sub languages *Logical-KARL (L-KARL)* and *Procedural-KARL (P-KARL)* to model the KADS layers, and combines techniques from logic programming and deductive databases.

The Domain-Layer Primitives

KARL uses L-KARL to describe the domain layer. It provides predicates, classes, class hierarchies, single- and set-valued attributes with domain and range restrictions, and multiple attribute inheritance for modelling terminological domain knowledge. The derivation of new object denotations can be expressed by functions. KARL uses O-logic [KiW93] and F-logic [KLW93] as a model for the integration of object-orientation in a logical framework.The Horn clauses of L-KARL are extended by stratified negation (cf. [Prz88]) and a richer logical language is provided for formulating constraints (necessary conditions for concepts and relationships).

The Inference-Layer Primitives

KARL distinguishes elementary and composite inference actions. An *elementary inference action* is described by a set of Horn clauses with stratified negation, again in L-KARL.

A subset of an inference structure can be arranged to form a *composed inference action*. These composed inference actions can be used as elementary elements to specify further inference steps, which gives a mechanism for hierarchical decomposition at the inference layer.[6]

KARL distinguishes three types of knowledge roles. *Views* define an upward translation from the domain layer to the inference layer (giving read-access). *Terminators* define a downward translation from the inference layer to the domain layer (giving write-access). *Stores* provide the input or output of inference actions. Whereas views and terminators are used to link a domain layer with a generic inference layer, stores are used to model the dataflow dependencies between inference actions.

The Task-Layer Primitives

KARL uses the logical language *Procedural-KARL (KARL)* at the task-layer. It is a variant of dynamic logic ([Har84], [Koz90]) and therefore allows the declarative description of control flow. Extended by additional syntactical sugar it can be used in a similar way to procedural languages.

The primitive programs correspond to *calling an inference action*, and atomic formulae indicate whether knowledge roles contain elements of a given class. Such primitive programs and atomic formulae can be arranged into sequence, loop, and alternative. Programs may be combined to named *subtasks*, similar to procedures in programming languages. Subtasks must correspond to composed inference actions at the inference layer.

The Primitives Connecting the Layers

Views and terminators constitute KARL's mechanism to connect domain and inference layer. Horn clauses with stratified negation of L-KARL are used to translate domain-expressions (or

6. This idea is identical to levelled dataflow diagrams in Structured Analysis [You89].

combinations of them) onto the contents of knowledge roles or support knowledge on the inference layer. A knowledge role need not correspond to a predefined domain concept and an inference action need not correspond to a predefined domain relationship. Several domain expressions can be combined to form a new expression by means of a view definition (cf. [ElN89]).

The connection between inference and task layer is formed by (i) the fact that primitive programs correspond to primitive inference actions; (ii) composed programs must correspond to composed inference actions, and (iii) state variables of dynamic logic store the contents of knowledge roles. A one-to-one correspondence must exist between the decomposition hierarchies at both layers.

3.5.2 The Formal and Operational View

The logical language L-KARL used to describe the domain, the inference layer, and their connection has a Herbrand model semantics [Llo87]. KARL allows stratified negation under the closed-world assumption using the minimal (i.e. perfect) Herbrand model as semantics [Prz88]. Constraints check this model for correctness. In contrast to Prolog, the evaluation of these clauses is set-oriented [Ull88]: not one but all instantiations of a predicate are computed.

The procedural knowledge is represented by P-KARL. It is a variant of Dynamic Logic which has a modal semantics [Koz90]. The integration of the modal semantics of the task-layer and the Herbrand models of L-KARL is as follows: the models of L-KARL are used to define an interpretation for a P-KARL language, i.e. the perfect Herbrand model of a set of clauses is used to interpret a function symbol occurring in value assignments in P-KARL.

An interpreter for KARL has been implemented which includes a debugger. A hypertext-based tool has been implemented which makes it possible to structure verbal protocols in a so-called semiformal Hyper model [NeM93]. This semiformal model can be used to built up a formal specification with KARL by refining the informal specification.

KARL provides graphical representations of most modelling primitives: a kind of Enhanced-Entity-Relationship (EER) diagrams for the domain layer, a kind of levelled dataflow diagrams for the inference layer, and a kind of program flow diagrams for the task layer. All three graphical representations include hierarchical refinement to allow them to represent the system at different levels of granularity.

3.5.3 Applications and Discussion

KARL has been applied in more than ten case-studies (e.g. [AFL91a], [AFL92b], [Ang92], [LHS92], [KFG92], [FEM93], and [LFA93]).

KARLS's major aim is to combine both formalisation *and* operationalization of models of expertise. This aim calls for a delicate balance of choices in the language. An example of this is the restriction on Horn logic with stratified negation and the choice for the perfect Herbrand model as semantics. This might be the correct choice for the declarative semantics, but it is less clear that the decision to regard the whole model (chosen to ensure that the procedural interpretation is guaranteed to compute the perfect Herbrand model) is also the most appropriate choice from an operational point of view.

A further concern in this same area is the fact that the set-oriented evaluation strategy restricts

KARL to predicates with a finite extension. Whether or not this formal restriction will be a problem remains to be seen.

A positive result of the concern of the KARL-designers with the practical usability of their language is the introduction of hierarchical refinement constructs at all levels in their language. No doubt this is a useful addition to the KADS framework.

3.6 $(ML)^2$

$(ML)^2$ ([BaA92b], [HaB92], [AHS93]) is a language for formalizing KADS models of expertise. It was developed in the course of the ESPRIT Projects 3178 “REFLECT” and 5248 “KADS-II” and a bilateral research project of the Netherlands Energy Research Foundation ECN and the University of Amsterdam. Although a subset of $(ML)^2$ can be operationalized to allow explorative prototyping it is mainly introduced as a formalization language.

3.6.1 The Epistemological View

$(ML)^2$ uses logic as its foundation, and combines different logical formalisms to model the structure of a KADS model.

The Domain-Layer Primitives

The sublanguage of $(ML)^2$ used to model a domain layer is order-sorted first-order logic extended by modularisation. Instances are modelled by constants, and sorts can be used to model classes of such constants. Sorts can be arranged in an is-a hierarchy. Relationships between concepts are modelled by predicates. Attributes of concepts are modelled by functions. Arbitrary first-order theories can be used to specify the defined relationships. The specification of a domain layer can be divided into several modules. Such a module or theory defines a signature (i.e. sorts, constants, functions, and predicates) and defines axioms (i.e. logical formulae). These modules, i.e. subtheories, can be combined by a union operator.

The Inference-Layer Primitives

In $(ML)^2$ every inference action and every knowledge role is described by a theory similar to domain layer theories. An inference action is described by an implication whose conclusion is a predicate naming the action and its input-and output-terms. The premises of such an implication describes the relation that must hold between input- and output-terms. Figure figure 6 illustrates an inference layer modelled in $(ML)^2$.

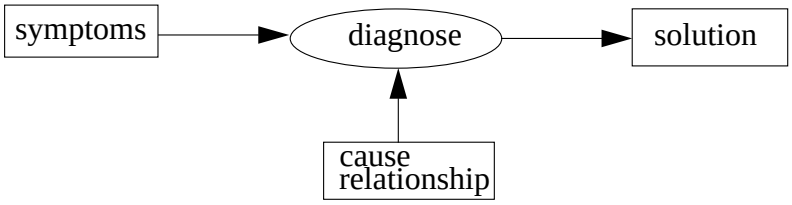

Figure 6. An inference action in $(ML)^2$

The inference action *diagnose* diagnoses a *solution* using *symptoms* and a *cause relationship* as input. This inference action *diagnose* is modelled by a predicate:

diagnose(symptoms(X), cause_relationship(symptoms(X),Y),

solution(Y))

For a description of an inference it is possible to use a set of clauses with this predicate as their conclusions. The whole description of the inference action *diagnose* is:

theory *diagnose*
use symptoms, cause_relationships
signature
predicates $\text{pia}_{\text{diagnose}}$
variables X, Y
axioms
$\text{pia}_{\text{diagnose}}$(
symptoms(X),
cause_relationship(symptoms(X), Y),
solution(Y)
) ←
$\text{input}_{\text{symptoms}}$(symptoms(X)) ∧
$\text{input}_{\text{cause\_relationship}}$(cause_relationship(symptoms(X), Z) ∧
Y=Z.

A theory which describes a knowledge role defines a translation of domain layer expressions to terms used at the inference layer (see below). Thus, each role and action is modelled by a separate theory, and a theory describing a primitive inference action imports all theories of its input- and output-knowledge roles.

The Task-Layer Primitives

Quantified-dynamic logic ([Har84], [Koz90]) is used to specify dynamic control at the task layer. Every predicate specifying an inference action at the inference layer together with the test operator "?" is regarded as an elementary program statement and the knowledge roles are used as input and output parameter of such programs. For every such elementary program a history variable is defined which stores the input-output pairs for every execution step. The key idea is to nondeterministically choose a value binding of a logical variable by the test operator and store this value in a state variable:

inference_action_predicate(input,output)?;
history_variable := append(history_variable | (input,output))

Four types of task-layer operations are available for each inference action: checking if a given Input/Output-pair (I/O-pair) has already been computed, checking if uncomputed I/O-pairs still exist, actually computing a new I/O-pair (and storing it in the corresponding history variable), and checking if a given I/O-relation holds (without storing it). These primitive programs and predicates can be combined using sequence, non-deterministic iteration and non-deterministic choice. These combinations are rich enough to model more standard constructions like deterministic iteration and conditional statements.

The Primitives Connecting the Layers

The inference and task layers are modelled as a meta-language of the domain layer. This meta-relation allows the inference-layer to specify properties of relations over domain-layer formulae without resorting to second order logic. Object- and meta-language are connected by a naming relation and reflection rules.

At the inference layer a lift operation is defined for every knowledge role which is connected to the domain layer. The lift operator defines a naming relation by mapping expressions of the domain layer to variable-free terms at the inference layer. This lift-operator is defined as a rewrite-rule system that translate domain-layer sentences into inference-layer terms.

This naming relation is used in three reflective predicates which can inspect or alter the domain

layer from the inference layer. One reflective predicate inspects the axioms of the domain layer, a second inspects the set of consequences of the domain layer,[7] and a third predicate extends the set of axioms at the domain layer. Therefore, the first two predicates are used to read from the domain layer whereas the last one can be used to write on it.

The connection between inference and task layer is realised by introducing primitive programs and predicates for each inference action, and by the history variables which represent the contents of the knowledge roles.

3.6.2 The Formal and Operational View

$(ML)^2$ provides three kinds of logical languages to describe models of expertise:

- *Order-sorted first-order logic* extended by modularisation at the domain layer. This language is equivalent to normal first-order logic, and therefore semi-decidable. Of all the languages discussed so far $(ML)^2$, is the first where the expressive power of finite axiom sets exceeds Turing completeness.
- *Meta-logic* at the inference layer.
- *Quantified-dynamic logic* at the task layer. It is a modal extension of first-order logic, which can be given a possible-world semantics.

With their combination of semi-decidable formalisms, $(ML)^2$ models cannot expected to be fully executable. [THR91] discusses a prototype interpreter for a subset of $(ML)^2$, where the main restriction is to reduce the domain layer to Horn clauses, and to interpret negation as negation as failure. More recent work [AbH92] describes a second interpreter, this time for full $(ML)^2$. It uses a full first-order theorem prover, and guarantees neither efficiency nor termination. [BaA92a] describes TheME, which supports the formalization of models of expertise using $(ML)^2$. It allows a graphical representation of $(ML)^2$ expressions and, for example, checks for syntactical correctness.

3.6.3 Applications and Discussion

$(ML)^2$ has been applied in a dozen formalizations of models of expertise [HaB92], e.g., for making predictions about physical systems by means of qualitative reasoning [BRW90]. The problem-solving methods *Cover and Differentiate* and *Heuristic Classification* [SWA92] are formalized in $(ML)^2$. A scheduling task with simple search heuristics has been specified in [BHA93].

$(ML)^2$ applies different types of logic for formalizing a model of expertise. First-order logic, meta-logic and quantified-dynamic logic are means for the different layers. $(ML)^2$ was the only language which provided a logical description for the control knowledge at the task layer (this solution has later been adopted by KARL). The application of meta-logic to represent knowledge at the inference layer comes close to the meaning of this layer.

A disadvantage of $(ML)^2$ is that it is only semi-decidable and it is not possible to evaluate it efficiently. It is no surprise that the operationalized subset of $(ML)^2$ is restricted to Horn logic and has a different semantics (i.e., negation as failure). In general, $(ML)^2$ is designed mainly as a formalization language allowing the declarative description of models of expertise. But it

7. In general, this predicate is only semidecidable.

becomes very difficult even for experts in logic to really understand the meaning of formally specified knowledge, especially if the models become very large and operationalizing a subset of the language under a different semantics cannot really overcome this problem.

3.7 QIL

QIL is a language developed at the University of Nottingham. Although QIL is originally intended as a model for multi-agent systems, where multiple agents can have beliefs that evolve over time, [ARS92] shows how QIL can be used to describe KADS models.

Of the languages discussed in this paper, QIL and $K_{BS}SF$ are the only languages that were originally designed for general knowledge-representation purposes, whereas the other languages were specifically developed to represent KADS-models.

3.7.1 The Epistemological View

QIL is based on a multi-modal logic, combining epistemic and temporal modalities. First-order logic is used to represent the declarative aspects of expertise, and the dynamic nature of problem solving is represented as the evolution of the beliefs of an agent over time.

KADS models are represented as the beliefs of an agent. The beliefs of the agent change when inference actions are applied. The application of such inference actions is modelled by logical deduction, while strategic knowledge is represented by planning.

The Domain-Layer Primitives

QIL uses a *reified*[8] first-order logical language for the representation of domain layer knowledge. All first-order formula are encoded as terms which are represented as arguments to the special predicate *domain-theory*. Thus, the formula

domain-theory(respiratory-tract, implies(airways-obstruction, less-than(fev1,70%)))

encodes what would more traditionally be written as the formula

airways-obstruction $\rightarrow$ fev1 < 70%

in a theory called *respiratory-tract*.

No proof-theory is defined in QIL for function symbols as *implies*, and consequently, no inference can be done at the domain layer.

The Inference-Layer Primitives

In QIL, the inference layer is represented in the same language as the domain layer: it is also a first order language, and all symbols for the domain layer could be referred to in the inference layer.

Knowlege roles are modelled as arguments to the special predicate *current-hypothesis*, and inference actions are modelled as implications between occurrences of this predicate. Such implications can also refer to the relations that are constructed out of domain-layer formula by means of the special predicate *knowledge-theory*. This predicate represents the link between domain and inference layer (see below). For example, as in figure 6 an inference action called *diagnose*, which uses domain relations to map the knowledge-role *symptoms* to the

8. In a *reified* language, formulae from another language are treated as terms.

knowledge-role *solution* would be represented as follows:

$\forall$ X $\forall$ Y (current-hypothesis(solution(Y)) $\leftarrow$
current-hypothesis(symptoms(X)) $\wedge$
knowledge-theory(causal,symptoms(X),Y)).

The Task-Layer and Strategy-Layer Primitives

QIL does not represent fixed task-decompositions (the usual contents of a KADS task-layer) but instead constructs sequences of subtasks dynamically, as part of the reasoning process. Such planning of subtask-sequences is typically seen as the role of the KADS strategy layer.

This planning process is specified by planning rules, encoded in QIL using the temporal modality. For instance, the planning rule

i-future(sub-task$_1$,instantiated(solution)) $\leftarrow$ always(instantiated(symptom))

states that whenever the knowledge role *symptom* is instantiated, then immediately after execution of *sub-task$_1$*, the knowledge role *solution* will be instantiated. General rules for the construction of plans specify how the above specific planning rules can be used to form a plan that will satisfy the goal. For instance, the modal formula:

always(bel(kbs, $\forall$ X $\forall$ ACT:
execute(ACT) $\leftarrow$
task-goal(X) $\wedge$ i-future(ACT, true(X))))

states that any action *ACT* that leads to the satisfaction of goal *X* will be executed. The expression *bel(kbs,*ϕ*)* states that the KBS-agent believes ϕ. Thus, planning rules are represented as beliefs of an agent about the future. Rules as this one can specify which actions should be executed, but QIL does not currently include facilities for choosing among actions if more than one action is executable at the same time.

The Primitives Connecting the Layers

The domain and inference layer in QIL are represented in the same language. The rules that link domain and inference layer ensure that the domain specific terms used in the domain layer segment of the language are translated to the generic terms used in the inference layer segment of the language. This translation is done via implications between the predicate *domain-theory* (whose arguments represent (reified forms of) the domain formula) and the predicate *knowledge-theory*, whose arguments are names of knowledge roles etc. For example:

$\forall$ X $\forall$ Y (knowledge-theory(causal,symptoms(X),Y)
$\leftarrow$ domain-theory(respiratory-tract,implies(X,Y)))

This way of connecting domain and inference layer is strongly reminiscent of the view-mechanism in KARL and the lift definitions in $(ML)^2$.

The connection between inference and task layer in QIL consists of rules that define the special predicate *instantiated* which is used in the planning rules, as shown above. For each knowledge role at the inference layer, the task layer contains a constant with the same name, and the predicate *instantiated* holds for such constant exactly when a value for that knowledge role has been derived at the inference layer (indicated by the predicate *current-hypothesis*. For example, for the knowledge role *observation*, there is an implication of the form:

instantiated(symptoms) $\leftarrow$ $\exists$X current-hypothesis(symptoms(X))

3.7.2 The Formal and Operational View

QIL is based on a first-order multi-modal logical language, containing both epistemic and temporal modalities. Domain and inference layer are modelled in a classical first-order logic with function symbols. Task and strategy layer are modelled in a modal language, containing the epistemic modality *believe*, and the temporal modalities *future* (at some point in the future), *always* (at all time points), and the nonstandard operator *i-future*. This latter operator takes two arguments, an action and a proposition, and its intended meaning is that the proposition is true immediately after performing the action. QIL is given a standard possible-world semantics, using the system K for the epistemic modality, and a forward branching model of time for the temporal modality. For quantification, a constant universe of discourse is assumed in QIL. The combination of the two modalities is completely orthogonal, and QIL makes no assumptions about persistency or otherwise of belief over time.

Like any modal logic, QIL can be given a reified definition inside a first-order language, and [ARS92] have investigated several resolution based theorem-proving methods for automating proofs in such reified logics. Such theorem provers will in general be neither efficient nor complete.

3.7.3 Applications and Discussion

[ARS92] gives a formulation of heuristic classification in QIL and [AKS93] gives a model of hierarchical skeletal planning.

An interesting aspect of QIL is that it shows how a non-KADS-specific language can be configured to capture KADS models. In QIL this is done by giving prescriptions for how the various theories should be organised, and by prescribing special predicates and implications to be used in these theories to model the different aspects of KADS models, in particular the predicates *domain-theory*, *knowledge-theory*, *current-hypothesis*, *instantiated*, the implications that model inference actions, and those that model the links between domain, inference and task layer.

The use of a temporal logic to model the procedural aspects of KADS models is similar in spirit to the use of dynamic logic in $(ML)^2$ and KARL, although rather different in technical detail.

QIL is one of the few languages that tries to dynamically construct task-decompositions in a KADS model of expertise.

An omission in the strategic layer of QIL would seem to be the inability to specify preferences over multiple plans that achieve the same goal. The choice among such alternative plans is currently relegated to the specific implementation of the theorem-proving machinery that would process a QIL model. It would be desirable if such knowledge could instead be explicitly included in QIL formulations of KADS models.

3.8 $K_{BS}SF$

$K_{BS}SF$[9] is developed by PTT Research in Groningen, The Netherlands as the language for describing conceptual models in the VITAL project [JSV91]. Thus, like QIL, $K_{BS}SF$ is not originally intended to describe KADS models, but the conceptual models of VITAL are very

9. Knowledge Based Systems Specification Language, pronounced as “KSF”.

close in spirit to those in KADS, and [JoS92], [VJS93] show how $K_{BS}SF$ can be used to represent KADS models.

$K_{BS}SF$ is primarily aimed at formalising conceptual models, and thus shares this aim with languages like $(ML)^2$ and QIL. $K_{BS}SF$ has many of its roots in software engineering, notably the techniques of algebraic specification languages.

3.8.1 The Epistemological View

$K_{BS}SF$ uses techniques from algebraic specifications for data-specification, a logical formalism for knowledge specification, and an imperative procedural language for control specification.

The Domain-Layer Primitives

The domain layer in $K_{BS}SF$ is modelled by a set of so-called Dmodules (Data-modules). A Dmodule consists of sort, function and predicate declarations which define the language used by the module; a set of equations which define equality among terms; theories which are sets of first-order sentences defining the truth conditions on the predicates, and signature types, which define the signatures used in those theories. A domain theory similar to the one give for QIL above would be:

```
Dmodule respiratory_tract
        import Numbers
        predicates >: Numbers, Numbers, airways_obstruction
        functions fev1: -> Numbers
        sigtypes T1 = [sorts Numbers, predicates > airways_obstruction]
        theories T1_th = {airways_obstruction -> fev1 > 70}
```

The Inference-Layer Primitives

To model knowledge roles, $K_{BS}SF$ uses parameterized signature types. These parameterized signature types define formal sorts and formal predicates that will be used in the inference actions to describe the inference steps. Thus, generic terms in KADS inference structures like *hypothesis* correspond in $K_{BS}SF$ to a single signature type, typically defining a single formal predicate name to model the knowledge role, plus the required formal sorts for the arguments of the formal predicate. The connection between a generic inference layer and a specific domain layer is then obtained by binding the formal predicate and sorts from the inference layer to actual predicates and its sorts from the domain layer.

Inference actions in $K_{BS}SF$ are modelled by a Bmodule (Behaviour module). A Bmodule consists of a name, a set of typed input and output parameters, a set of local variables, and a behaviour description in a procedural language containing assignments, loops, conditionals and a mechanism for calling other task as subroutines. Furthermore, special operations like intersection, union and selection can be performed on either the axiom sets of theories or their deductive closures.

A inference action thus corresponds to a Bmodule with an input and output parameter for each input and output knowledge role, plus a task-body that specifies a procedure for computing the output values from the input values.

The Task-Layer Primitives

$K_{BS}SF$ does not distinguish between task and inference layer, and simply regards inference actions as primitive subtasks (i.e. those tasks that do not call other subtasks). Thus, a task structure is represented in the same way as the inference action, namely as a set of Bmodules.

The Primitives Connecting the Layers

As stated above, a generic inference layer (in terms of parameterized signature types) is connected to a specific domain layer by binding the formal sorts and predicates of the parameterized signature types (representing the knowledge roles) to actual predicates and sorts of the domain layer. Thus, the connection between domain and inference layer is obtained through parameter binding.

Since no distinction is made between inference and task layer, no connecting primitives between these layers are required.

3.8.2 The Formal and Operational View

The algebraic components of $K_{BS}SF$ (used for data representation) consist of terms and equations over these terms. These can be given the usual initial algebra semantics, meaning that terms are equal only when they can be proven equal by the equations. Elements of a sort then consist of all the equivalence sets of terms in that sort, and only these.

The domain knowledge in $K_{BS}SF$ is represented by theories. Since these are sets of first order sentences, the usual Tarskian model theoretic semantics applies.

For the imperative language in task bodies no declarative formal semantics is given, but the properties of such simple imperative languages are well known [Dij76].

Given the very powerful primitives in $K_{BS}SF$ (algebraic rewrite rules, deductive closures of full first-order theories) $K_{BS}SF$ could not be supported by an efficient interpreter, although interpreters for subsets of the language would seem feasible. Tools like type- and syntax-checkers are currently under development.

3.8.3 Applications and Discussion

[JoS92] gives a small formulation of a model for heuristic classification, while in [VJS93] a simple time scheduling task is formalised.

The most striking thing about $K_{BS}SF$ is its strong exploitation of parameterization to formulate generic models. The use of such parameterization for the link between domain and inference layer seems an elegant and compact yet powerful solution.

$K_{BS}SF$ is not specifically designed with KADS model of expertise in mind, and therefore departs in some places from this model: the lack of separation between inference and task layer is a sharp example of this. This lack of distinction is particularly important since a procedural language is used at this combined level, which leaves the KADS inference layer without a declarative description and mixes control knowledge into the inference structure. This is in contrast to many of the other languages. $K_{BS}SF$ also joins the ranks of KARL and MoMo by providing graphical representations for almost all the language elements, so that visual presentation and manipulation of $K_{BS}SF$ models is possible.

With its powerful representational mechanisms, $K_{BS}SF$ would seem more suited for formal

analysis of a model than for simulation or prototyping. In this sense, it is close in motivation to a language like $(ML)^2$, although the actual formal constructions differ considerably between the two languages. The current lack of a formal and complete semantics of $K_{BS}SF$ is therefore an important shortcoming.

Chapter 4
How can these different languages be compared?

In this section, we will apply the comparison criteria from section 2 to the languages that have been described in section 3. Rather than simply *discovering* differences between the languages through applying the comparison criteria, we are also interested in *explaining* these differences. One of the main explanatory forces behind the differences between the various languages will be the different *aims* of the various languages. We can distinguish three classes of languages:[10]

- Languages which aim to *operationalize* models of expertise, like OMOS, MODEL-K and MoMo. These languages add the advantages of prototyping, i.e. the evaluation by a running program, to the modelling process.
- Languages which aim to *formalize* models of expertise like $(ML)^2$, QIL and $K_{BS}SF$. These languages allow a precise formulation of a model of expertise and enable checks for consistency, redundancy, and correctness by verification techniques.
- Languages which attempt *both*, like FORKADS and KARL. These languages aim at integrating the advantages of formal and operational languages.

Our comparison criteria in section 2 were organised in 3 groups: epistemological, formal and operational. Clearly, the epistemological criteria apply to all languages. The formal criteria are mainly relevant to languages of the last two classes mentioned above, while the operational criteria are concerned with languages of the first and last class. Beyond simply limiting the applicability of our criteria to the various classes of languages, this division between operational and formalisation languages (or both) will also serve to explain many of the differences to be found in the comparison of these languages in this section.

4.1 Comparison from the Operational View

Not all languages *aim* to operationalize models of expertise (criterion O1). For example, $K_{BS}SF$ is not discussed as an operational language and is therefore not included in this part of the comparison. QIL and $(ML)^2$ are executable in principle, requiring a theorem prover. FORKADS, KARL, MODEL-K, MoMo, and OMOS are languages which aim to *operationalize* models of expertise.

Concerning the *computational paradigm* (crit. O2), we can use the operational vs. formalisation distinction among the languages. We see that all of the languages aiming at formalisation have chosen logic as their main computational paradigm. Other paradigms (e.g. functional) are chosen by the operationalization languages.

Significant differences exist concerning the *operational semantics* of the languages (crit. O3):

10. A similar point of view is taken in [Sch92].

- All the operational languages have an operational semantics (OMOS, MODEL-K, MoMO, FORKADS and KARL)
- Of the two languages that aim to combine operationalization and formalisation (FORKADS and KARL) only KARL has both an operational and a declarative semantics, and these coincide. Furthermore, it is the only language whose declarative semantics is efficiently computable.
- Of the languages aiming at formalisation (FORKADS, KARL, $(ML)^2$, QIL, $K_{BS}SF$) all but FORKADS have a declarative semantics, but the pure formalisation languages ($(ML)^2$, QIL, $K_{BS}SF$) either lack an operational semantics ($K_{BS}SF$), or have an operational semantics which is ineffective (QIL) or that is only a subset of the declarative semantics ($(ML)^2$).

The question whether a language is a means to operationalize a model of expertise is highly related to its *computational power* (crit O4). All languages which aim to operationalize models of expertise have a computational power which is less or equal to Turing completeness (OMOS is the only language which has a computational power less than Turing completeness). QIL and $(ML)^2$ (which could be executed by use of a theorem prover) have at least the expressive power of first-order logic.

MODEL-K and OMOS are based on BABYLON and MoMo is based on Lisp (CLOS). *Efficiency* (crit. O5) is not discussed for these languages but it is clear that the implementation of inference actions in BABYLON or Lisp and the very limited manner of inference actions in OMOS should allow an efficient evaluation. As a consequence, specifications in MODEL-K or MoMo could contain a significant amount of symbol level control knowledge, i.e. knowledge about the implementation of single inference steps by a programming language (see [Sch92] for the distinction of symbol level and knowledge level control).[11] FORKADS bypasses this by a set of predefined and implemented primitive inference actions. The main difference between a formal and operational language like KARL and an operationalization language like MODEL-K is how they model inference action bodies. In KARL primitive inference actions bodies are described declaratively whereas in MODEL-K they are implemented algorithmically. Therefore, an operational knowledge specification in KARL does not require specification of symbol level control. As a consequence, the evaluation of declaratively described inference actions in KARL is less *efficient* than the evaluation of inference actions which are implemented by efficient algorithms and data structures as is possible in MODEL-K or FORKADS by its procedural attachment. To bypass this problem, KARL restricted the logical language to stratified Horn logic with finite models[12] and applies bottom-up evaluation techniques developed in the domain of deductive data bases. The languages $(ML)^2$ and QIL, which have the expressive power of first-order logic, require a theorem-prover as operationalization, which in general allows neither an effective (because of

11. At the knowledge level there is a description of the domain knowledge and the used problem-solving method which is required by an agent to solve the problem effectively and efficiently. At the symbol level there is a description of efficient algorithmic solutions and data structures for implementing an efficient computer program, i.e. a very specific agent.

12. It is not the perfect models that are finite (because, for example, KARL contains all Integers) but the extensions of user-defined predicates, which are computed by a fixpoint operator.

the incompleteness of proof techniques for first-order logic) nor an efficient evaluation.[13]

1. Comparison at the operational level

Language	Executable (O1)	Computat. Par. (O2)	Semantics (O3)	Expressiveness (O4)	Effectiveness /Efficiency (O4,O5)
OMOS	Yes	Procedural	Operational	less than Turing complete	Yes/Yes
MODEL-K	Yes	Functional	Operational	Turing complete	Yes/Yes
MoMo	Yes	Functional	Operational	Turing complete	Yes/Yes
FORKADS	Yes	Logical	Operational	Turing complete	Yes/?[a]
KARL	Yes	Logical	Operational and declarative	Turing complete	Yes/No
$(ML)^2$	Not effectively	Logical	Declarative	more than Turing complete	No/No
QIL	Not effectively	Logical	Declarative	more than Turing complete	No/No
$K_{BS}SF$	No	Logical	Declarative	more than Turing complete	No/No

a. Efficiency of FORKADS is not discussed in the literature.

Table 1 summarizes this section. It shows the almost complete correspondence between the aim of the language on the one hand, and the values of the various operational properties on the other.

4.2 Comparison from the Formal View

Again, we have to ask for the *aim* of the different languages (criterion F1). $K_{BS}SF$, QIL, and $(ML)^2$ are developed for formalizing models of expertise. FORKADS and KARL both aim to operationalize and to formalize models of expertise. MODEL-K, MoMo, and OMOS are not discussed as formalization languages and are therefore not included in this section.

All of these languages use logic as their fundamental *mathematical basis* (crit. F2), either full predicate logic, or restricted to Horn clauses. $K_{BS}SF$ also uses algebraic structures.

The essence of a formal language is its *formal semantics* (crit F3). QIL, $(ML)^2$ and KARL fulfil this requirement by having a declarative semantics. At the domain and inference layers, QIL and $(ML)^2$ apply the standard Tarskian semantics for first order predicate logic, while KARL uses the minimal, i.e. perfect, model semantics from logic programming and deductive

13. In [THR91]] a prototype interpreter $Si(ML)^2$ for a subset of $(ML)^2$ is discussed. The main restriction is the restriction on Horn logic. This subset of $(ML)^2$ is quite similar to KARL.

databases. Strikingly, at the task-layer all of these languages use a Kripke-type semantics for a modal logic (either temporal or dynamic). Some important differences are found in the solutions for expressing *dynamic behaviour* in a declarative framework (crit. F2):

- FORKADS and $K_{BS}SF$ do not have a declarative semantics for dynamic behaviour.
- $(ML)^2$ and KARL use dynamic logic to model behaviour. Dynamic logic [Koz90] was developed to describe procedural programs declaratively. Programs are interpreted by binary relations which express the connection of input and output of a program. Whereas $(ML)^2$ stores the whole traces, i.e. all computed values, of a problem-solving process as an increasing list of values in the variables, KARL stores the most recently computed values, only.
- QIL uses temporal logic to model behaviour. Derived results are indexed by time-points, and the dynamics of the computation are modelled by the sequence of these time-points at which results are derived.

It is no surprise, that the two languages FORKADS and KARL which aim to operationalize and formalize models of expertise provide the least *expressive power* of all formalization languages (crit. F4). FORKADS and KARL are Turing-complete, whereas the languages $K_{BS}SF$, $(ML)^2$, and QIL are at least omega-complete (∞-complete), i.e. finite axioms sets in these languages have at least the expressive power of finite axioms sets in first-order logic. It should be remarked as a critical comment that none of all these languages specify their exact degree of expressiveness.

Table 2 summarizes this section. It shows a remarkable degree of similarity between the formal foundations of all these languages.

2. Comparison at the formal level

Language	Semantics of static knowledge (F2,F3)	Semantics of dynamic knowledge (F2,F3)	Expressiveness (F4)
FORKADS	Tarskian models	No declarative semantics	Turing complete
KARL	Perfect Herbrand models	Kripke models	Turing complete
$(ML)^2$	Tarskian models	Kripke models	At least first-order logic
QIL	Tarskian models	Kripke models	At least first-order logic
$K_{BS}SF$	Tarskian models and initial algebras	no declarative semantics	At least first-order logic

4.3 Comparison at the Epistemological level

In this section we will compare the languages on the basis of their different ways of representing KADS models. It will turn out that there is rather widespread agreement about the primitives required for the domain and task layers, and we will sketch these briefly. Most interesting for the purposes of our comparison are the mechanisms used for representing the inference layer. This is where we find the greatest variety of representations and interpretations.

4.3.1 Domain layer

Apparently, much agreement exists over the *representation primitives* required at the domain layer (criterion D1). The dominant choice here is some type of declarative formalism, based on either logic or frames and hierarchies. An exception is MoMo, which deliberately leaves the choice of the domain language open. Among the different languages, we find much variation in syntactic form, but this is often only a superficial matter from a technical point of view. From a conceptual point of view, one can distinguish between languages like QIL or $(ML)^2$ which propose a uniform representation (e.g. logic extended by typing and modularisation) and languages like MODEL-K or KARL which propose representations with a greater number of different modelling primitives. This allows the expression of epistemologically different types of knowledge by different modelling primitives. In contrast with the original KADS literature, none of these languages has chosen a KL-ONE like representation of concepts, instances, properties, values and relations, although all the languages are rich enough to express these notions.

Because all these languages use some form of declarative formalism (logic or frames), it is indeed possible to represent domain knowledge *independent from its use* and free from control knowledge (crit. D2). Some languages explicitly distinguish between *data and knowledge* at the domain layer (crit. D3), e.g. KARL and $K_{BS}SF$, while other languages do not enforce this distinction (although it is still possible to make such a distinction, e.g. in $(ML)^2$ or QIL). This distinction is indeed useful (it is also integrated in the new KADS-II framework [WVS93] as the case-model), and it is therefore commendable to provide syntactic support for this distinction within the language.

The choice for the language used at the domain layer immediately imposes a number of assumptions about what *types of knowledge* can and cannot be modelled at the domain layer (crit. D4). In particular, none of the languages include mechanisms for dealing with uncertain or time-dependent domain knowledge (although a number of languages could be easily adapted to deal with such extensions). A more deeply rooted assumption (at least for the logic-based languages) concerns the monotonicity of domain layer knowledge. The inheritance mechanism of some of the frame-based languages can to some extent deal with non-monotonic knowledge (by overriding of defaults)

A significant difference among the languages at the domain layer is perhaps the various mechanisms that are employed for *modularisation* of domain-layer constructs (crit. D6). These vary from modules (subtheories) with a simple union operation to sophisticated parameterization mechanisms as in $K_{BS}SF$. All the languages besides QIL allow some form of *inferencing* at the domain layer (crit. D5): either deduction in the logic-based languages, or inheritance in the frame-based languages.

Table 3 summarises the choices for the languages used at the domain layer. The table only

3. Comparison at the domain layer

Language	Language for domain layer (D1)	knowledge/data distinction (D3)	modularisation (D6)
OMOS	Frames, hierarchy, predicates	no	no
MODEL-K	Frames, hierarchy, predicates	no	no
MoMo	Open	no	no

3. Comparison at the domain layer

Language	Language for domain layer (D1)	knowledge/data distinction (D3)	modularisation (D6)
FORKADS	Sorted-logic	no	no
KARL	Frames + Logic	yes	modules with import/export
$(ML)^2$	Sorted-logic	no	subtheories and union
QIL	Logic	no	clause-index arguments
$K_{BS}SF$	Logic and algebraic equality theory	yes	parameterized theories

shows those criteria where there is some variation among the languages. There is widespread agreement about the choices at this layer, with the only distinction being that the operational languages use frames and the formalisation languages use logic as the main basis for this layer.

4.3.2 Inference Layer

Of all the layers of a KADS model, this layer accounts for the largest differences between the various languages, not only in the syntactic constructions, but also in the interpretation of the constituents of a KADS inference layer.

One of the central components of an inference layer are the *primitive inference actions*, the others being knowledge roles. Rather widespread agreement exists on the choice of language to represent these inference actions (criterion I1), and once again the distinction between operational and formalisation language is crucial: table 4 shows that all operational languages use a functional representation of an inference action, and all formalisation languages use a logical representation.

Besides the inference actions, the *knowledge roles* are the other main modelling primitive at the inference layer. Much more consensus exists on the representation of these knowledge roles (crit. I2). Table 4 shows that all languages represent these as either sets or as lists of the input and output elements of inference actions.

The large differences between the various languages in their interpretation of a KADS inference layer are clearly illustrated by the relation between domain and inference layer in the various languages (the *domain view*, crit. I3). A crucial aspect of the inference layer in a KADS model is that it should abstract from the domain layer (crit. I4). In this way, the inference layer can describe a generic inference model, which still has access to the contents of the domain knowledge, but that does not depend on its specific contents. To achieve this aim, almost every language introduces some mechanism to express a mapping between domain and inference layer (see table 4). However, the details of these mechanisms differ greatly, and no two languages use the same mechanism. No agreement exists on the relative merits of these mechanisms. An exception to the use of a mapping mechanism is OMOS, which uses direct references from the inference to the domain layer (thereby violating the requirements of a

KADS inference layer). In some other languages such direct reference to domain knowledge is in fact possible (QIL, KARL), but would be regarded as abuse of the language.

The different representations of inference actions have consequences for another important requirement of a KADS inference layer, namely that the inference actions be free from *internal control* (crit I5). This requirement holds for the formalisation languages which use a declarative logical formulation of the I/O-relation of an inference action, but does not hold for the operational languages OMOS, MODEL-K, and MoMo, because the functional or procedural representation makes a much greater algorithmic commitment. The description of the model of expertise is mixed with implementational aspects. Clearly, this larger amount of control knowledge inside an inference action is motivated by the operational aim of these languages.

Another aspect of control at the inference layer is less clear, and concerns *control among the inference actions* (such as the sequence they should be executed in, crit. I6). KADS states that such control among inference actions should not be expressed at the inference layer. In some languages (e.g. MoMo), it is possible to express such control, although this is not the intended use of this language. In other languages (e.g $(ML)^2$), it is impossible to express such control, but the data-dependencies between inference actions (as expressed in the inference structure) is taken as a restriction on the possible execution orders of these inference actions. Yet other languages (e.g. KARL) leave even this representation of control among inference actions out of the inference layer.

The network of knowledge roles and inference actions at the inference layer (the inference structure) represents the data-dependencies between inference actions, and KADS states that the only *communication between inference actions* is through connecting knowledge roles (crit. I7). Although unintended, in many languages it is in fact possible to have other communications between inference actions (e.g. via shared access to the domain layer). Again, we see a gap between the intended and the possible uses of a language.

The original KADS framework proposed a small set of *fixed inferences* that should be used as building blocks for inference structures (crit. I8). It is therefore rather surprising that most of the languages actually allow an open-ended set of inferences to be specified by the user. The only exceptions are OMOS and FORKADS, which do restrict the set of possible inferences. OMOS provides a template for inference actions which must be parameterized and FORKADS provides a set of predefined inference actions. Until now, it is not clear whether this set is complete, or how large a complete set of elementary inference actions must be. Recent work in the KADS-II project [Abe93] suggests that an open-ended set of schemes for inference actions might be more appropriate than a closed set of fixed inferences.[14]

The choice of representation of an inference action (as apparently determined by the operational or formalisation aim of a language) also accounts for the degree of *non-determinism* of an inference action (crit. I9). In KADS, an inference action need not be functional, in the sense that a single input value can result in a number of different output values. Non-determinism at the inference layer is defined as the possibility to compute one of these output values for a single input value, without specifying which particular output value should be computed. Table 4 shows that none of the operational languages can express this type of non-determinism (since they all use a functional representation of an inference action). All the formalisation languages (who use a logical representation) can express this. An exception to this pattern is KARL (a language aiming at both operationalization and formalisation). KARL is deterministic since it always computes the entire extension of an inference action (i.e. all input/output pairs). It is clear that the removal of non-determinism is

14. See also [Cla92a] who proposes a library of model construction operators instead of libraries of problem-solving methods or inference actions.

advantageous for operational languages.

Since the original KADS framework did not provide a mechanism for *hierarchical decomposition* at the inference layer (an inference layer is a flat graph of inference actions and knowledge roles), not many of the languages incorporate such a decomposition mechanism at this layer (crit. I10). Exceptions are KARL (which does provide a decomposition mechanism at the inference layer), and $K_{BS}SF$, which does not distinguish between task- and inference-layer, and is thus able to exploit its decomposition mechanism for the task layer at the inference layer.

Looking at table 4, which lists the most varying criteria for the inference layer, we can say in summary that the distinction between operational and formalisation languages can account for many, although not all of the difference between the languages at the inference layer. The representation of inference actions, the amount of control knowledge inside an inference action and the possibility to represent non-deterministic inference actions can all be explained by looking at the operational or formal aim of the language. Other differences however seem to form genuinely different interpretations of the inference layer in the KADS model of expertise, such as the need for a decomposition mechanism, a set of predefined inference actions, and the description of control among the inference actions. A remarkable variety of mechanisms has been employed to obtain an abstraction between domain and inference layer, which all more or less agree on the overall aim of this mapping, but differ greatly on their technical details.

4. Comparison at the inference layer

Language	Language for inference actions (I1)	Repre-sentation of knowledge roles (I2)	Mapping to domain layer (I3)	Control inside inference actions (I5)	Predefined set of inferences (I8)	Non determ. (I9)	Decom-position mecha-nism (I10)
OMOS	Functional	Set	Direct naming	yes	Yes	No	No
MODEL-K	Functional	List	Parameter binding	yes	No	No	No
MoMo	Functional	Multi-set	LISP code	yes	No	No	No
FORKADS	Logical	Set	Type-graph morphism	no	Yes	Yes	No
KARL	Logical	Set	Stratified clauses	no	No	No	Yes
$(ML)^2$	Logical	List	Rewrite rules	no	No	Yes	No
QIL	Logical	Set	Horn clauses	no	No	Yes	No
$K_{BS}SF$	Logical	Set	Parameter binding	no	No	Yes	Yes

4.3.3 Task layer

As with the domain layer, and unlike the inference layer, much agreement again exists

concerning the primitives to be used on the task layer, at least when looking at the languages superficially. The dominant choice for the *representation primitives* (criterion T1) is a procedural language with classical constructions from programming languages, such as sequence, loop, branching, and some *decomposition mechanism* like subroutines (crit. T2). QIL is an exception since it computes task-structures dynamically.

5. Comparison at the task layer

Languages	Conditional expressions (T4)	Non-determinism (T6)
OMOS	Did most recent inference action change the content of a knowledge role?	No
MODEL-K	Arbitrary LISP predicates on knowledge roles	No
MoMo	Cardinality of knowledge roles plus success/failure of inference actions	Yes
FORKADS	Arbitrary test on knowledge roles	No
KARL	Does a knowledge role contain an element of a specific class or not?	No
$(ML)^2$	Arbitrary predicates on knowledge roles	Yes
QIL	---	---
$K_{BS}SF$	Arbitrary predicates on knowledge roles	Yes

When looking in slightly more detail, we should consider issues like the *mechanism used to represent the state of the problem-solving process* (crit. T3), what types of *conditional expressions* can be used to inspect the contents of knowledge roles (crit. T4), and whether *specifications can be non-deterministic* (crit. T6). The dominant view on how to represent the *state* is clearly by the current contents of the knowledge roles. The only exceptions is $(ML)^2$, which has a richer representation, namely not only the current but also *all* the past contents of the knowledge roles. Concerning the type of *conditional expressions* allowed on the task layer, we see that all languages restrict this to conditional expressions on the contents of knowledge roles, but are otherwise rather liberal in the type of expressions that they allow. With the exception of OMOS, MoMo, and KARL they allow arbitrary tests on the contents of the knowledge roles. These liberal possibilities are in conflict with more recent KADS literature which has proposed that such expressions should perhaps be limited to a fixed set of rather simple tests on knowledge roles regarded as sets or lists, such as membership, cardinality, element-selection, etc.

All of the languages maintain a strict *separation* between task and domain layer (crit. T5).

As a final aspect, we see that only three languages allow for *non-deterministic* specifications of the control knowledge (that is to say: to leave certain control choices unspecified). All other language enforce a deterministic control regime.

Table 5 summarises these distinctions. No important structural differences are found among the languages at the task-layer, and both operational and formalisation languages have made similar choices at this layer.

4.3.4 Strategic Layer

MODEL-K is the only language that makes a serious attempt at providing constructs for the strategic layer. The REFLECT project [HWB92]) has suggested that the strategic layer should be regarded as a full KADS model in its own right, whose domain layer consists of the first 3 layers of the original model, and it has worked out this suggestion in MODEL-K. [HaB92] speculatively regards the strategic layer in $(ML)^2$ as constructive theorem proving in dynamic logic. QIL's strategy layer as theorem-proving in a temporal logic is very similar in spirit. However, no general agreement on the use of the strategic layer exists conceptually, and this is the main reason that no detailed language constructs have been introduced for this layer.

4.3.5 The connections between the layers

As already discussed in the previous section, a large variety of mechanisms is employed to achieve *the connections between inference and domain layer* (criterion C1), without much agreement on the relative merits of these mechanisms. Every language with the exception of OMOS includes such a mechanism. In some languages, it is strictly speaking possible to circumvent the connection mechanism, and to refer directly to the domain layer expressions from the inference layer. OMOS is the only language in which this direct reference is the only possibility. An important property of the relation between domain and inference layer in KADS models is the fact that each knowledge role element is only allowed to refer to a single domain-layer expression. In other words, the mapping from inference to domain layer must be functional, but need not be injective. The languages differ in the extent to which they enforce this constraint. For instance, the parameter binding mechanism from $K_{BS}SF$ guarantees that the mapping from inference to domain layer is functional. In $(ML)^2$, it is possible to specify a non-functional mapping, but whether a specific mapping is functional or not is a decidable property. A language like MoMo, with its very free-form specification of the mapping gives no guarantee on the properties of the mapping.

Whereas a multitude of complex mechanisms is used to relate domain and inference layer, *the relation between task and inference layer* (crit. C2) is much simpler. All languages employ a one-to-one relation between primitive tasks and inference actions for the relation between these two layers, and most often this mapping is realised by simply identifying the names of primitive tasks at the task layer with the inference actions at the inference layer. The presence of a hierarchical decomposition mechanism at the inference layer (in KARL and $K_{BS}SF$) somewhat complicates the relation with the task layer. In KARL, the entire subtask-decomposition must have a one-to-one relation with the decomposition at the inference layer. In $K_{BS}SF$, the two layers have simply been merged into one language, removing the need for a mapping mechanism.

One of the reasons for separating the model of expertise into different layers is to allow these layers to be independently reusable (crit. C3). Almost all languages allow the reuse of a single domain layer by multiple inference layers, and, vice versa, the reuse of a single inference layer for multiple domain layers. The exception is OMOS, whose inference layer makes direct reference to domain layer terminology, so that its inference layer cannot be reused. Although the reuse between domain and inference layer is possible in principle in almost all languages, not many examples of such reuse have been reported: some inference structures have been reused in multiple domains, but we have found no report on the reuse of a domain layer by multiple inference layers. Because all task layers make direct reference to the primitive inferences at the inference layer, task layers can in general not be reused for different inference

layers. The extent to which an inference layer can be reused by different task layers directly depends on the amount of control knowledge encoded in the inference layer in a particular language.

Conclusion

In this concluding section of the paper, we will discuss the following topics: Have these formal languages clarified the informal KADS models? How do these knowledge-engineering languages relate to similar work in software engineering? How can we combine these different languages? What themes for future work can we distil from all this?

Clarification of informal KADS models

One of the motivations for work on formal languages is to clarify the informal notions which inspired them. This process has indeed taken place. By developing more precise languages to express KADS model of expertise, multiple interpretations of these models were revealed. An example of this are the multiple interpretations of inference structures, and different properties of the connections between domain and inference layer. A second example of this is that in the informal KADS literature, it was claimed that the domain layer "can be viewed as a declarative theory of the domain, and adding a simple deductive capability would enable a system in theory to solve all problems solvable by the theory" [WSB92]. More detailed analysis with the formal languages described in this paper showed this not to be the cases (e.g. imagine a deductive causal theory that is used abductively for diagnostic reasoning). Furthermore, the development of these languages has shown the inadequacy of some aspects of KADS models, since almost every language deviated in these aspects from the original KADS literature. Example of this are the use of KL-ONE at the domain layer, and the notion of a fixed set of inferences at the inference layer.

Comparison with software engineering and information systems development

One reaction to the so-called *software crises* in the late sixties was the development of informal, formal, and operational specification languages in the domain of software engineering (SE) and later by the information systems development (ISD) community. In [IEEE77] and in [ACM82] a survey on informal and operational specification techniques is given. In [ThD90], [Zav91] the experiences with some of these approaches after a decade of applications is reported. Currently, there exists a huge number of these languages.[15] Originally, two mainstreams of research were the development of informal or operational specification techniques. More recently, the development of formal specification techniques, i.e. specification languages with a declarative semantics, which also have a long tradition, has become the most important line of research. *EREA* [DHR91], the short-cut for Entity, Relation, Attribute, and Event, the *Modal Action Logic (MAL)* [FiP87], *OBJ* [Shu89], the *Requirements Modelling Language (RML)* [GBM86], and *TROLL* [HJS93], *VDM*, the short-cut for *Vienna Development Methods*, and *Z* (cf. [BHL90] for VDM and Z) are some examples for formal specification languages in SE and ISD. Current lines of research are, for example, type-checking of specifications and the development of mechanizable proof theories based on the formal semantics of the languages [AGM93], [BFL93]. Although this is a very rough survey, it illustrates that to a some extent research in knowledge engineering (KE) reinvents the wheel.

15. In [AFL92a] a comparison of KARL with the specification language of the INCOME project [LNO89], which is used to specify information systems, can be found.

This uncomfortable situation has been changed to some extent in the last years because the number of publications has increased which relate the work in KE to work which has already been done in the domain of SE and ISD ([AFS90], [JSV91], [Sch92], [FAL93]).

One point where the languages from KE may have an edge of those from SE is that the former are strongly based on prescriptive conceptual models, such as the KADS model of expertise for the languages discussed in this paper. A model in these languages is much closer to an informal or semiformal description than models in low-level (i.e., general-purpose) languages like Z, which can be used to describe arbitrary programs via finite set theory. KE languages use the fact that they are used to describe a specific class of software artifacts. A lot of approaches in SE like Structured Analysis and Object-Oriented Analysis are based on semiformal methods only. Vice versa, most formal approaches do not aiming on integrating formal models with informal or semiformal specifications. This does not only concern the definition of graphical representations for the modelling primitives of a formal language but touch the question how a description of a system at the conceptual level does correspond with its description as a partial function. The tight integration of conceptual and mathematical description techniques is something that could be learned by SE from KE. Admittedly this is more easy in the case of KE, because its restricted class of software artefacts which are modelled.[16]

Combining the languages: the happy language family

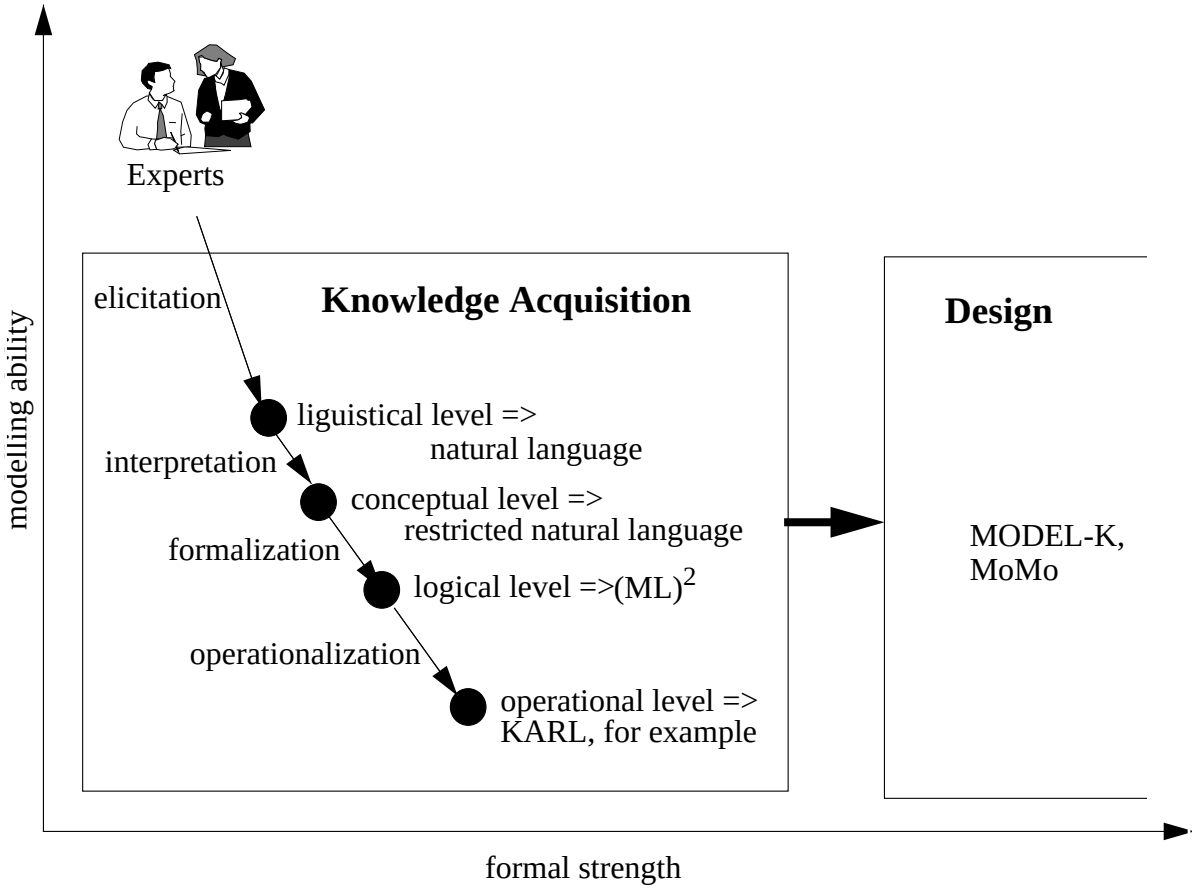

Figure 7. The happy family of the different languages

16. By the way, [FAL93] shows how KARL can be used to formalize and operationalize descriptions techniques of Structured Analysis [You89]. Therefore, the achieved results in KE can be generalized.

As we have seen in the preceding section, many of the differences between the languages do not arise from different solutions to the same goal, but instead from the different goals that the languages have. Formal and operational languages are complementary, rather than contradictory. This clearly opens the possibility of combining these languages in different stages of the knowledge engineering process (see figure 7). For instance, first a language like $(ML)^2$ could be used to formalise a model of expertise, since for the translation of informally specified expertise into a formal representation, it is helpful to have as much expressive power as possible. A language like KARL can be used in a second step to make the model of expertise executable. Finally, a language like MODEL-K could be used to increase the efficiency by implementing the bodies of the inference actions efficiently (instead of describing them declaratively, as done in KARL). This process of translating between these languages can support the transition from specification to design and implementation without destroying the conceptual structure of the modelled expertise.

Future work

From the analysis of the languages in this paper, three areas for future work in this field become apparent.

The first area where further work is needed is *applications*. Very few large-scale applications of any of these languages has been reported. As a result, the claim of reusability of the generic interpretation models expressed in these languages can not be validated. A point closely related to this is the need for libraries of predefined models which are expressed in these languages.

A second area where further work is required is *support* for the use of these languages. The specifications expressed in these languages quickly become large and complicated, and automated support in the form of syntax- and type-checkers and dedicated editors is indispensable. Much research in this area has already been done in the context of traditional software engineering specification languages, and should be applied to the knowledge engineering languages discussed here. Another form of support that is badly needed are *guidelines* for the use of these languages. In all of the languages discussed in this paper, there is a large difference between what *can* syntactically be written down, and what *should* be written down if we want to respect the constraints of KADS models. Since the intended use of the languages can not always be easily characterised syntactically, a further set of guidelines are required on this intended use. Most languages give no such guidelines beyond the informal text in the papers describing the languages. An early attempt at more systematic guidelines for $(ML)^2$ is the unpublished [Abe92], so much work remains to be done in this area.

A final area of further work is on the *formal semantics* of the languages. Even those languages that give some form of formal semantics (KARL, $K_{BS}SF$, $(ML)^2$) do so either only for components of the language without describing how this component-wise semantics should be combined ($K_{BS}SF$, $(ML)^2$), or use a nonstandard integration of the components of the semantics which deserves further study (KARL).

Acknowledgement

We thank all participants of the workshops on languages for KADS models (at GMD, Bonn in May 1992 and at the University of Karlsruhe in May 1993) for helpful and fruitful discussions: Stuart Aitken (University of Nottingham); Manfred Aben and Peter Terpstra (University of Amsterdam); Hans Akkermans and John Balder (ECN, Petten); Juergen Angele, Joachim Geidel, Dieter Landes, Susanne Neubert and Rudi Studer (University of Karlsruhe); Frances Brazier, Jan Treur and Mark Willems (Free University, Amsterdam); Mikael Eriksson and Per Kreuger (SICS, Sweden); Mihai Barbuceanu (Institute of Informatics, Bucharest); Willem Jonker, Jan Willem Spee and Linda in't Veld (PTT

Groningen); Werner Karbach, Marc Linster, Angi and Hans Voss (GMD Bonn); Thomas Wetter (IBM Heidelberg).

References

[Abe92] M. Aben: Guidelines for the Formal Specification of KADS Models of Expertise. In [BaA92a].

[Abe93] M. Aben: Formally Specifying Reusable Knowledge Model Components. In *Knowledge Acquisition Journal*, vol 5, no 2, 1993.

[AbH92] M. Aben and F. van Harmelen, *Design and Implementation of Si (ML)^2 $_{2.0}$* , technical report KADS-II/T1.2/SP/UvA/030/2.0, SWI, University of Amsterdam, October 1992

[ACM82] *ACM SIGSOFT Software Engineering Notes*, vol 7, no 5, 1982.

[AFL91a] J. Angele, D. Fensel, D. Landes, and R. Studer: An Assignment Problem in Sisyphus - No Problem with KARL. In [Lin92c].

[AFL92a] J. Angele, D. Fensel, and D. Landes: Two Languages to Do the Same? In *Proceedings of the 2nd Workshop Informationssysteme und Künstliche Intelligenz*, February 24-26, 1992, Ulm, Informatik-Fachberichte, no 303, Springer-Verlag, Berlin, 1992.

[AFL92b] J. Angele, D. Fensel und D. Landes: An Executable Model at the Knowledge Level for the Office-Assignment Task. In [Lin92d].

[AFL93] J. Angele, D. Fensel, D. Landes, S. Neubert, and R. Studer: Model-based and Incremental Knowledge Engineering: The MIKE Approach. In J. Cuena (ed.), *Knowledge Oriented Software Design, IFIP Transactions A-27*, North Holland, Amsterdam, 1993..

[AFS90] J. Angele, D. Fensel, and R. Studer: Applying Software Engineering Methods and Techniques to Knowledge Engineering. In D. Ehrenberg et.al. (eds.), *Wissensbasierte Systeme in der Betriebswirtschaft, Reihe betriebliche Informations- und Kommunikationssysteme, no 15*, Erich Schmidt Verlag, Berlin, 1990.

[AFS94] J. Angele, D. Fensel, and R. Studer: The Model of Expertise in KARL. In *Proceedings of the 2nd World Congress on Expert Systems*, Lisbon/Estoril, Portugal, January 10-14, 1994.

[AGM93] D. J. Andrews, J. F. Groote, C. A. Middelburg (eds.): *Preliminary Proceedings of the International Workshop on Semantics of Specification Languages SoSL*, Utrecht, The Netherlands, October 25-27, 1993.

[AHS93] H. Akkermans, F. van Harmelen, G. Schreiber, and B. Wielinga: A Formalisation of Knowledge-Level Models for Knowledge Acquisition. In *International Journal of Intelligent Systems, Special Issue on Knowledge Acquisition*, no 2, vol 8, 1993.

[AKS93] S. Aitken, O. Kühn, N. Shadbolt, F. Schmalhofer: A Conceptual Model of Hierarchical Skeletal Planning and its Formalization. In *Proceedings of the 3rd KADS Meeting*, Munich, March 8-9, 1993.

[Alf90] M. Alford: SREM at the age of eight; the distributed computing design system. In [ThD90], pp. 392-402.

[Ang92] J. Angele: Cover and Differentiate Remodeled in KARL. In [BaK92].

[Ang93] J. Angele: Operationalisierung des Modells der Expertise mit KARL (Operationalization of a Model of Expertise with KARL), Ph. D. thesis, University of Karlsruhe, 1993 (in German).

[ARS92] S. Aitken, H. Reichgelt, N. Shadbolt: Representing KADS models in QIL, AI Group, University of Nottingham, Working Paper WP-006, 1992.

[BaA92a] J. Balder and H. Akkermans: TheMe: An Environment for Building Formal KADS-II Models of Expertise. In *AI Communications*, vol 5, no 3, September 1992.

[BaA92b] J. Balder and H. Akkermans (eds.): Formal Methods For Knowledge Modelling in the CommonKADS Methodology: A Compilation, report ECN-C-92-080, Netherlands Energy Research Foundation ECN, ZG Petten, The Netherlands, December 1992.

[BaK92] C. Bauer and W. Karbach (eds.): *Interpretation Models for KADS - Proceedings of the 2nd KADS User Meeting (KUM´92)*, Muenich, February 17-18, 1992, GMD report no. 212, 1992.

[Bar93] M. Barbuceanu: Models: Towards Integrated Knowledge Modeling Environments. In *Knowledge Acquisition*, vol 5, no 3, 1993.

[Bee90] C. Beeri: A formal approach to object-oriented databases. In *Data and Knowledge Engineering*, vol 5, no 4, 1990.

[BFL93] J. C. Bicarregui, J. S. Fitzgerald, P. A. Lindsay, R. Moore, B. Ritchie: *Proof in VDM: A Practitioner´s Guide*, Springer Verlag, Berlin, 1993.

[BHA93] J. Balder, F.v. Harmelen, and M. Aben: A KADS/(ML)^2 Model of a Scheduling Task. In [TrW93].

[BHL90] D. Bjørner, C. A. R. Hoare, and H. Langmaack (eds.): *VDM´90. VDM and Z - Formal Methods in Software Development*, Lecture Notes in Computer Science, no 428, Springer-Verlag, Berlin, 1990.

[Bra79] R. J. Brachman: On the Epistemological Status of Semantic Networks. In N. V. Findler (eds.), *Associative Networks: Representation and Use of Knowledge by Computers*, Academic Press, New

York, 1979.

[BRW90] B. Bredeweg, M. Reinders, and B. Wielinga: GARP: A Unified Approach to Qualitative Reasoning, report VF-memo 117, University of Amsterdam, 1990.

[BüW92] S. Bürsner and Th. Wetter: An Operational KADS Modelling Language and Tool Support for its Application. In *Proceedings of the 7th Banff Knowledge Acquisition for Knowledge-Based System Workshop (KAW´92)*, Banff, Canada, October 11-16, 1992.

[ChJ93] B. Chandrasekaran and T.R. Johnson: Generic Tasks and Task Structures: History, Critique and New Directions. In [DKS93].

[Cla85] W.J. Clancey: Heuristic Classification. In *Artificial Intelligence*, vol 27, 1985.

[Cla92a] W. J. Clancey: Model Construction Operators. In *Artificial Intelligence*, vol 53, no 1, 1992.

[CPV89] T. Christaller, F. di Primo, and A. Voß: *Die KI-Werkbank BABYLON* (The KI-Tool BABYLON), Assison Wesley, Bonn, 1989 (in German).

[DHR91] E. Dubois, J. Hagelstein, and A. Rifaut: A Formal Language for the Requirements Engineering of Computer Systems. In A. Thayse (ed.), *From Natural Language Processing to Logic for Expert Systems*, John Wiley & Sons, Chichester, 1991.

[Dij76] E.W. Dijkstra: *A Discipline of Programming*, Engelwood Cliffs, N.J., Prentice-Hall, 1976.

[DKS93] J.-M. David, J.-P. Krivine, and R. Simmons (eds.): *Second Generation Expert Systems*, Springer-Verlag, Berlin, 1993.

[DKV92] U. Drouven, W. Karbach, and Angi Voß: Solving the Office-Allocation Task in Reflective MODEL-K. In [Lin92d].

[ElN89] R. Elmasri and S.B. Navathe: *Fundamentals of Database Systems*, The Benjamin/Cummings Publishing Company, Houston, 1989.

[FAL91] D. Fensel, J. Angele, and D. Landes: KARL: A Knowledge Acquisition and Representation Language. In *Proceedings of Expert Systems and their Applications, 11th International Workshop, Conference Tools, Techniques & Methods*, May 27-31, Avignon, 1991.

[FAL93] D. Fensel, J. Angele, D. Landes, and R. Studer: Giving Structured Analysis Techniques a Formal and Operational Semantics with KARL. In H. Züllighoven et al. (eds.), *Requirements Engineering ´93: Prototyping, German Chapter of the ACM Berichte*, no 41, Teubner Verlag, Stuttgart, 1993.

[FEM93] D. Fensel, H. Eriksson, M. A. Musen, and R. Studer: Description and Formalization of Problem-Solving Methods for Reusability: A Case Study. In *Complement Proceedings of the European Knowledge Acquisition Workshop (EKAW´93)*, Toulouse, France, September 6-10, 1993.

[Fen93a] D. Fensel: The Reconciliation of Symbol and Knowledge Level, research report, Instituts für Angewandte Informatik und Formale Beschreibungsverfahren, University of Karlsruhe, no 266, 1993.

[Fen93b] D. Fensel: The Knowledge Acquisition and Representation Language KARL, Ph. D. thesis, University of Karlsruhe, 1993.

[FiP87] A. Finkelstein and C. Potts: Building Formal Specifications Using “Structured Common Sense”. In *Proceedings of th 4th International Workshop on Software Specification and Design*, Monterey, California, April, 1987.

[Flo84] C. Floyd: A Systematic Look at Prototyping. In R. Budde et al. (eds.), *Approaches to Prototyping*, Springer-Verlag, Berlin, 1984.

[GBM86] S. J. Greenspan, A. Borgida, and J. Mylopoulos: A Requirements Modeling Language and its Logic. In *Information Systems*, vol 11, no 1, 1986.

[Gei92] J. Geidel: An Environment for Modelling and Solving Optimisation Problems. In *Proceddings of the 2nd IFIP WG 7.6. - Conference on Optimization-Based Computer Aided Modelling and Design*, Schloß Dagstuhl, Germany, 28th September - 1st October, 1992.

[GrB92] P. de Greef and J. A. Breuker: Analysing System-User Cooperation in KADS. In *Knowledge Acquisition*, vol 4, no 1, 1992. See also J. Breuka and P. de Greef: Modelling System-User Cooperation in KADS. In [SWB93].

[HaB92] F. v. Harmelen and J. Balder: $(ML)^2$: A Formal Language for KADS Conceptual Models. In *Knowledge Acquisition*, vol 4, no 1, 1992. See also F. v. Harmelen and J. Balder: $(ML)^2$: A Formal Language for KADS Models of Expertise. In [SWB93].

[Har84] D. Harel: Dynamic Logic. In D. Gabbay and F. Guenthner (eds.), *Handbook of Philosophical Logic, Vol. II: Extensions of Classical Logic*, Reidel, Dordrecht, The Netherlands, 1984.

[HeR91] O. Herzog and C.-R. Rollinger (eds.): *Text Understanding in LILOG*, Lecture Notes in Artificial Intelligence, no 546, Springer-Verlag, Berlin, 1991.

[HJS93] T. Hartmann, R. Jungclaus, and G. Saake: Spezifikation von Informationssystemen als Objektsysteme: Das TROLL-Projekt (Spezification of Information Systems as Object Systems: The TROLL-Project), *Emisa Forum*, no 1, 1993 (in German).

[HWB92] F. van Harmelen, B. Wielinga, B. Bredeweg, G. Schreiber, W. Karbach, M. Reinders, A. Voß, J. M.

Akkermans, and B. Bartsch-Spörl, and E. Vinkhuyzen: Knowledge-level Reflection. In B. Le Pape et al. (eds.), *Enhancing the Knowledge-Engineering Process - Contributions from ESPRIT*, Elsevier Science Publ., B. V., Amsterdam, 1992.

[IEEE77] IEEE Transactions on Software Engineering, vol 3, no 1, 1977.

[JoS92] W. Jonker and J.W. Spee: Yet Another Formalisation of KADS Conceptual Models. In *Proceedings of the 6th European Knowledge Acquisition for Knowledge-Based Systems Workshop (EKAW-92)*, May 18-22, Heidelberg/Kaiserslautern, T. Wetter et al. (eds.), *Current Developments in Knowledge Acquisition*, Lecture Notes in Artificial Intelligence, no 599, Springer-Verlag, Berlin, 1992.

[JSV91] W. Jonker, J.W. Spee, L. in ´t Veld, and M. Koopman: Formal Approaches Towards Design in SE and Their Role in KBS Design. In *Proceedings of the IJCAI´91 Workshop on Software Engineering for Knowledge-Based Systems*, Sydney, Australia, August 24th, 1991.

[Kar93] W. Karbach: MODEL-K: Modellierung und Operationalisierung von Selbsteinschätzung und -Steuerung durch Reflexion und Metawissen, Ph. D. thesis, University of Bielefeld, Germany, 1993 (in German).

[KaV92] W. Karbach and A. Voß: Reflecting About Expert Systems in MODEL-K. In *Proceedings of Expert Systems and their Applications, 12th International Workshop, vol 1 (Scientific Conference)*, June 1-6, Avignon, 1992.

[KaV93] W. Karbach and A. Voß: MODEL-K For Prototyping and Strategic Reasoning at the Knowledge Level. In [DKS93].

[Kee89] S. E. Keene: *Object-Oriented Programming in Common Lisp*, Addison-Wesley, Reading, 1989.

[KFG92] R. Köppen, D. Fensel, and J. Geidel: Modelling the Selection of Scheduling Algorithms with KARL. In [BaK92].

[KiW93] M. Kifer and J. Wu: A Logic for Programming with Complex Objects. To appear in *Journal of Computer and Systems Science*, 1993.

[KLS91] O. Kühn, M. Linster, and G. Schmidt: Clamping, COKAM, KADS, and OMOS: The Construction and Operationalization of a KADS Conceptual Model. In *Proceedings of the 5th European Knowledge Acquisition Workshop EKAW'91*, Crieff, Scotland, May 20-24, 1991, M. Linster et al. (eds.), GMD-Studien, no 211, September 1992.

[KLW93] M. Kifer, G. Lausen, and J. Wu: Logical Foundations of Object-Oriented and Frame-Based Languages, technical report 93/06, Department of Computer Science, SUNY at Stony Brook, NY, April 1993. To appear in *Journal of the ACM*.

[KoT90] W. Kowalczyk and J. Treur: On the Use of a Formalized Generic Task Model in Knowledge Acquisition. In B. Wielinga et al. (eds.), *Current Trends in Knowledge Acquisition*, IOS Press, Amsterdam, 1990.

[Koz90] D. Kozen: Logics of Programs. In J. v. Leeuwen (ed.), *Handbook of Theoretical Computer Science*, Elsevier Science Publ., B. V., Amsterdam, 1990.

[KVS91] W. Karbach, A. Voß, R. Schuckey, and U. Drouven: MODEL-K: Prototyping at the Knowledge Level. In *Proceedings of Expert Systems and their Applications, 11th International Workshop, Conference Tools, Techniques & Methods*, May 27-31, Avignon, 1991.

[LFA93] D. Landes, D. Fensel, and J. Angele: Formalizing and Operationalizing a Design Task with KARL. In [TrW93].

[LHS92] D. Landes, D. Hackenberg, and T. Schweier: An Inference Structure for a Configuration Problem. In [BaK92].

[LiM92] M. Linster and M. Musen: The Inference Structure of K-ONCOCIN: Skeletal Plan Refinement. In [BaK92]. See also M. Linster and M. Musen: A KADS Conceptual Model of the ONCOCIN Task. In [SWB93].

[Lin91] M. Linster: Tackling the Office-Plan Problem with OMOS. In [Lin92c].

[Lin92a] M. Linster: Knowledge Acquisition Based on Explicit Methods of Problem Solving, Ph. D, thesis, University of Kaiserslautern, February 1992.

[Lin92b] M. Linster: Linking Modeling to Make Sense and Modeling to Implement Systems in an Operational Modeling Environment. In *Proceedings of the 6th European Knowledge Acquisition for Knowledge-Based Systems Workshop (EKAW-92)*, May 18-22, Heidelberg/Kaiserslautern, 1992, T. Wetter et al. (eds.), *Current Developments in Knowledge Acquisition*, Lecture Notes in Artificial Intelligence, no 599, Springer-Verlag, Berlin, 1992.

[Lin92c] M. Linster (ed.): Sisyphus ´91: Models of Problem Solving, Arbeitspapiere der GMD, no 630, March 1992.

[Lin92d] M. Linster (ed.): Sisyphus ´92: Models of Problem Solving, Arbeitspapiere der GMD, no 663, July 1992.

[Lin92e] M. Linster: Using the Operational Modelling Language OMOS to Tackle the Sisyphus´92 Office-Planning Problem. In [Lin92d].

[Lin93] M. Linster: Using OMOS to Represent KADS Conceptual Models. In [SWB93].

[LKV92] M. Linster, W. Karbach, A. Voß, and J. Walther: An Analysis of the Role of Operational Modelling Languages in the Development of Knowledge-Based Systems, *Proceedings of the 2nd Japanese Knowledge Acquisition for Knowledge-Based Systems Workshop (JKAW´1992)*, Hatoyama, Japan, November 9-13, 1992.

[Llo87] J.W. Lloyd: *Foundations of Logic Programming, 2nd Editon*, Springer-Verlag, Berlin, 1987.

[LNO89] G. Lausen, T. Nemeth, A. Oberweis, F. Schönthaler, and W. Stucky: The INCOME Approach for Conceptual Modelling and Prototyping of Information Systems. In *Proceedings of the 1st Nordic Conference on Advanced Systems Engineering CASE'89*, Stockholm, Sweden, May 9-11, 1989.

[LPT92] Izak van Langevelde, A. Philipsen, and J. Treur: A Compositional Architectures. In Proceedings of the 10th European Conference on AI (ECAI-92), Vienna, Austria, August 3-7, 1992.

[LPT93] Izak van Langevelde, A. Philipsen, and J. Treur: Formal Specification of Compositional Architecture for Simple Design Formally Specified in DESIRE. In [TrW93].

[Mar88] S. Marcus (ed.): *Automating Knowledge Acquisition for Experts Systems*, Kluwer Academic Publisher, Boston, 1988.

[MDK92] D. Marques, G. Dallemagne, G. Klinker, J. McDermott, and D. Tung: Easy Programming: Empowering People to Build Their Own Applications. In *IEEE Expert*, vol 7, no 3, 1992.

[Möl92] J.-U. Möller: Towards Declarative Programming in Conceptual Models. In *Proceedings of the 2nd Workshop Informationssysteme und Künstliche Intelligenz*, February 24-26, Ulm, 1992, Informatik Fachbericht, no. 303, Springer-Verlag, Berlin, 1992.

[Mus89] M. A. Musen: *Automated Generation of Model-Based Knowledge-Acquisition Tools*, Morgan Kaufmann Publisher, San Mateo, CA, 1989.

[NeM93] S. Neubert and F. Maurer: A Tool for Model Based Knowledge Engineering. In *Proceedings of the 13th International Conference AI, Expert Systems, Natural Language (Avignon´93)*, Mai 24-28, Avignon, 1993.

[New82] A. Newell: The Knowledge Level. In *Artificial Intelligence*, vol 18, 1982.

[PET92] A. R. Puerta , J. W. Edgar, S. W. Tu, and M. A. Musen: A Multiple-Method Knowledge-Acquisition Shell For The Automatic Generation of Knowledge-Acquisition Tools. In *Knowledge Acquisition*, vol 4, no 2, 1992.

[Prz88] T. C. Przymusinski: On the Declarative Semantics of Deductive Databases and Logic Programs. In J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufmann Publisher, Inc., Los Altos, CA, 1988.

[SAW89] G. Schreiber, H. Akkermans, and B. Wielinga: On Problems with the Knowledge Level Perspective. In *Proceedings of the 5th Knowledge Acquisition for Knowledge-Based Systems Workshop*, Banff, Canada, November, 1989.

[Sch92] G. Schreiber: *Pragmatics of the Knowledge Level*, Ph. D. Thesis, University of Amsterdam, 1992.

[Shu89] R. N. Shutt: A Rigorous Development Strategy Using the OBJ Specification Language and the MALPAS Program Analysis Tool. In *Proceedings of the 2nd European Software Engineering Conference ESEC´89*, Warwick, UK, September 11-15, 1989, *Lecture Notes in Computer Science*, no 387, Springer-Verlag, Berlin, 1989.

[Ste92] L. Steels: Reusability and Configuration of Applications by Non-Programmers, technical report VUB AI-Memo 92-4, Free University of Brussel, Brussels, 1992. See also L. Steels: The Componential Framework and its Role in Reusability. In [DKS93].

[SWA92] G. Schreiber, B. Wielinga, and H. Akkermans: Differentiating Problem Solving Methods. In *Proceedings of the 6th European Knowledge Acquisition for Knowledge-Based Systems Workshop (EKAW-92)*, May 18-22, Heidelberg/Kaiserslautern, 1992, T. Wetter et al. (eds.), *Current Developments in Knowledge Acquisition*, Lecture Notes in Artificial Intelligence, no 599, Springer-Verlag, Berlin, 1992.

[SWB93] G. Schreiber, B. Wielinga, and J. Breuka (eds.): *KADS. A Principled Approach to Knowledge-Based System Development*, Knowled-Based Systems, vol 11, Academic Press, London, 1993.

[ThD90] R. H. Thayer and M. Dorfman (eds.): *System and Software Requirements Engineering*, IEEE Computer Society Press, Washington, 1990.

[THR91] A. t. Teije, F.v. Harmelen, and M. Reinders: Si(ML)^2: A Prototype Interpreter for a Subset of (ML)^2, ESPRIT project P5248 KADS-II, report KADS-II/T1.2/TR/UvA/005/1.0, University of Amsterdam, 1991.

[TrW93] J. Treur and Th. Wetter (eds.): *Formal Specification of Complex Reasoning Systems*, Ellis Horwood, New York, 1993.

[Ull88] J. D. Ullman: *Principles of Database and Knowledge-Base Systems, vol I*, Computer Sciences Press, Rockville, Maryland, 1988.

[VJS93] L. in ´t Veld, Willem Jonker, and J. W. Spee: The Specification of Complex Reasoning Tasks in

K_{BS}SF. In [TrW93]

[VKS91] A. Voß, W. Karbach, R. Schuckey, and U. Drouven: The Office Planning Problem in MODEL-K. In [Lin92c].

[VoK93] A. Voß and W. Karbach: MODEL-K: Making KADS Run. In [SWB93].

[VoV93] H. Voss and A. Voss: Reuse-Oriented Knowledge Engineering with MoMo. In *Proceedings of the 5th International Conference on Software Engineering and Knowledge Engineering (SEKE´93)*, San Fransisco Bay, June 14-18, 1993.

[VVW93] A. Voß, H. Voß, J. Walther, and T. Hemman: Model-Driven Prototyping - Prototyping-Driven Modelling in Knowledge-Based System. In H. Züllighoven et al. (eds.), *Requirements Engineering ´93: Prototyping, German Chapter of the ACM Berichte*, no 41, Teubner Verlag, Stuttgart, 1993.

[Wet90] T. Wetter: First Order Logic Foundation of the KADS Conceptual Model. In B. Wielinga et al. (eds.), *Current Trends in Knowledge Acquisition*, IOS Press, Amsterdam, 1990.

[Wet92] T. Wetter: FORKADS: An Executable Language for the KADS Conceptual and Interpretation Models, Habilitationsschrift, University of Kaiserslautern, Germany, 1992.

[WeS91] T. Wetter and W. Schmidt: Formalization of the KADS Interpretation Models. In *Proceedings of the 8th Conference of the Society for the Study of Artificial Intelligence and Simulation of Behaviour (AISB91)*, Leeds, GB, April 16-19, 1991.

[WSB92] B.J. Wielinga, A.Th. Schreiber, and J.A. Breuker: KADS: A Modelling Approach to Knowledge Engineering. In *Knowledge Acquisition*, vol 4, no 1, 1992. Also appeared in [SWB93].

[WVS93] B. J. Wielinga, W. Van de Velde, A. Th. Schreiber, and J. M. Akkermans: Towards a Unification of Knowledge Modelling Approaches. In [DKS93].

[You89] E. Yourdon: *Modern Structured Analysis*, Prentice Hall, Englewood Cliffs, 1989.

[Zav91] P. Zave: An Insider´s Evaluation of PAISLey. In *IEEE Transactions on Software Engineering*, vol 17, no 3, 1991.